ZigBee PRO Smart Energy API User Guide

JN-UG-3059

Revision 3.4

22 April 2013

Contents

About this Manual

This manual provides an introduction to ZigBee Smart Energy (SE) and describes the use of the NXP ZigBee PRO Smart Energy Application Programming Interface (API) for the JN51xx wireless microcontroller. The manual contains both operational and reference information relating to the ZigBee PRO Smart Energy API, including descriptions of the C functions and associated resources (e.g. structures and enumerations).

Note: Clusters that are part of the ZigBee Cluster Library (ZCL) but used by the Smart Energy profile are detailed in the *ZCL User Guide (JN-UG-3077)*, which you should use in conjunction with this manual.

The API is designed for use with the NXP ZigBee PRO stack to develop wireless network applications based on the ZigBee Smart Energy application profile. For complementary information, refer to the following sources:

- Information on ZigBee PRO wireless networks is provided in the *ZigBee PRO Stack User Guide (JN-UG-3048),* available from **www.nxp.com/jennic**.
- The ZigBee SE profile is defined in the *ZigBee Smart Energy Profile Specification (075356)*, available from the ZigBee Alliance.

Organisation

This manual is divided into four parts:

- Part I: Concept and Development Information comprises four chapters:
  - Chapter 1 introduces the concept of Smart Energy (SE) and an SE network
  - Chapter 2 describes the ZigBee implementation of Smart Energy
  - Chapter 3 provides an overview of ZigBee PRO SE application development using NXP resources
  - Chapter 4 describes the essential aspects of coding a ZigBee PRO SE application using the NXP ZigBee PRO Smart Energy API
- Part II: Smart Energy Clusters comprises seven chapters:
  - Chapter 5 describes the SE-specific implementation of clusters from the ZigBee Cluster Library (ZCL) as well as the Over-the-Air Upgrade cluster
  - Chapter 6 details the Price cluster used in ZigBee PRO SE
  - Chapter 7 details the Messaging cluster used in ZigBee PRO SE
  - Chapter 8 details the Simple Metering cluster used in ZigBee PRO SE
  - Chapter 9 details the Demand-Response and Load Control cluster used in ZigBee PRO SE

- Chapter 10 details the Key Establishment cluster used in ZigBee PRO SE
- Chapter 11 details the Tunnelling cluster used in ZigBee PRO SE

- Part III: General Reference Information comprises two chapters:
  - Chapter 12 details the core functions of the Smart Energy API (initialisation function and device-specific endpoint registration functions)
  - Chapter 13 details the general (not cluster-specific) structures, enumerations and parameters used by the Smart Energy API
- Part IV: Appendices contains appendices which summarise the supported clusters and attributes, and describe how to set up custom endpoints.

Conventions

Files, folders, functions and parameter types are represented in **bold** type.

Function parameters are represented in *italics* type.

Code fragments are represented in the `Courier New` typeface.

This is a **Tip**. It indicates useful or practical information.

This is a **Note**. It highlights important additional information.

*This is a **Caution**. It warns of situations that may result in equipment malfunction or damage.*

Terminology

The following changes have been made by the ZigBee Alliance to ZigBee Smart Energy terminology:

- The Energy Service Portal (ESP) is now referred to as the Energy Service Interface (ESI). However, to maintain consistency with existing NXP software, the term ESP will continue to be used in this manual.
- The Simple Metering cluster is now referred to as the Metering cluster. However, to maintain consistency with existing NXP software, the term Simple Metering cluster will continue to be used in this manual.

Acronyms and Abbreviations

AMR	Automated Meter Reading
APDU	Application Protocol Data Unit
API	Application Programming Interface
DRLC	Demand-Response and Load Control
EPID	Extended PAN ID
ECDSA	Elliptic Curve Digital Signature Algorithm
ESI	Energy Service Interface
ESP	Energy Service Portal
HAN	Home Area Network
IDE	Integrated Development Environment
IPD	In-Premise Display
LCE	Load Control Event
NAN	Neighbourhood Area Network
OTA	Over-the-Air
PAN	Personal Area Network
PCT	Programmable Communicating Thermostat
PDU	Protocol Data Unit
SE	Smart Energy
TOU	Time Of Use
UTC	Co-ordinated Universal Time
ZCL	ZigBee Cluster Library

Related Documents

JN-UG-3077	ZigBee Cluster Library User Guide
JN-UG-3048	ZigBee PRO Stack User Guide
JN-UG-3075	JenOS User Guide
JN-UG-3064	SDK Installation and User Guide
JN-UG-3007	JN51xx Flash Programmer User Guide
JN-AN-1135	Smart Energy HAN Solutions Application Note
075356	ZigBee Smart Energy Profile Specification [from ZigBee Alliance]
075123	ZigBee Cluster Library Specification [from ZigBee Alliance]

Trademarks

All trademarks are the property of their respective owners.

Chip Compatibility

The software described in this manual can be used on the following NXP wireless microcontrollers:

- JN516x (currently only JN5168-001)
- JN5148-Z01 (limited distribution)

Where the described functionality is applicable to all the supported microcontrollers, the device may be referred to in this manual as the JN51xx.

Part I: Concept and Development Information

1. An Introduction to Smart Energy

The efficient and prudent use of energy is now a global pre-occupation, the over-riding motivation being environmental damage-limitation to offset climate change. Further, high energy prices and widespread ‘fuel poverty’ have heightened the need to re-evaluate our energy consumption. This has led to the concept of ‘Smart Energy’ (SE).

1.1 Philosophy of Smart Energy

Smart Energy is an approach to the clever use of energy, encompassing measures ranging from keeping consumers informed about their power consumption to the automated re-scheduling of power-hungry activities into off-peak/low-price periods. It is a methodology that is adopted and deployed by the energy provider or “utility” company but, for a successful implementation, needs to be equally embraced by their customers. The form of the supplied energy can be electricity or gas (and other resources, such as water, can also be incorporated into the scheme).

The objectives and principles of Smart Energy are detailed in the sub-sections below.

1.1.1 SE Objectives

The objectives of Smart Energy are to:

- minimise energy consumption (and therefore production) in order limit its environmental impact
- minimise energy costs for consumers
- smooth out energy demand to avoid peaks that put strain on energy production
- encourage customers to generate their own energy through clean sources (e.g. solar panels) and give them credit for it
- introduce a degree of automation, including Automated Meter Reading (AMR)

The operational principles for attaining the above objectives are described next, in Section 1.1.2.

1.1.2 SE Principles

The simplest Smart Energy system just incorporates ‘smart meters’ at the consumers’ premises, where these meters can be read remotely by the utility company - Automated Meter Reading (AMR). While such a system does not contribute towards the environmental objectives of Smart Energy, it allows the utility company to operate more efficiently, with resulting benefits for the consumer. A true Smart Energy system also includes other devices at the consumers’ premises, allowing energy and cost savings to be made in the following ways:

- **Real-time display of data:** The consumer is informed in real-time of their rate of energy (power) consumption and the associated cost. It may even be possible to report the power consumption of a particular device. This encourages the consumer to become more energy-conscious and allows them to make decisions about their energy usage - for example, they may decide to delay using their clothes dryer until a low-price period.
- **Control from utility company (demand-response):** Certain power-hungry devices or systems (e.g. air-conditioning system) at the consumers’ premises could be dynamically managed from the utility company according to the energy-supply conditions at the time. For example, if the demand for power is high, the utility company could request a device to switch off or reduce its power consumption. This is known as the demand-response feature.
- **Intelligent appliances:** Under the Smart Energy scheme, appliances could become commercially available that are able to receive communications from the utility company and modify their operation accordingly - for example, when switched on, a dish washer could automatically delay its start until it becomes aware that a low-price period has begun.
- **Exporting generated power:** Customers with the ability to generate their own electric power (from clean power sources such as solar panels and wind turbines) may be able to sell any surplus power to the national grid. A Smart Energy system can provide a means of measuring this power contribution and relaying the measurements to the utility company so that the customer’s account can be credited accordingly.

The Smart Energy devices needed to implement the above methods are described in Section 1.2.

1.2 Smart Energy Devices

The devices needed to implement Smart Energy at a customer's premises form a local network at the premises, sometimes referred to as a Home Area Network (HAN). This network is installed by the utility company and is linked to a larger network, the backhaul network, which allows communication with the utility company headquarters. The networking aspects of Smart Energy are described in Section 1.3, while here we describe the Smart Energy devices used to implement the principles and achieve the objectives detailed in Section 1.1.

In this manual, we are concerned with HANs implemented using the ZigBee PRO wireless network protocol. We will therefore adopt the device terminology used in the ZigBee Smart Energy specification. Note that the ZigBee definitions of these devices are covered in Section 2.4.

The Smart Energy devices are as follows:

- Energy Service Portal (ESP)
- Metering Device
- In-Premise Display (IPD)
- Programmable Communicating Thermostat (PCT)
- Load Control Device
- Smart Appliance
- Prepayment Terminal

These devices are described below.

Energy Service Portal (ESP)

The Energy Service Portal (ESP) is the device which connects the HAN to the backhaul network of the utility company. It therefore provides the entry and exit points for communications between the utility company and the HAN. Every Smart Energy HAN must have an ESP. Note that the ESP may be contained in the same physical unit as another SE device, such as the Metering Device, an In-Premise Display or a Programmable Communicating Thermostat (see below).

Metering Device

The Metering Device measures and records energy consumption, and other related data (the recorded data may also be mirrored on the ESP). It also allows this data to be remotely read by and/or periodically reported to the utility company.

In-Premise Display (IPD)

The In-Premise Display (IPD) contains some sort of visual display for relaying information to the consumer. IPDs can vary widely in their display methods and the type of information they can display:

- The type of information that an IPD may display includes current power consumption, historical energy consumption data, available pricing and important messages from the utility company.
- The display methods range from coloured lights (which indicate the current level of power consumption or price), through simple text-only screens to full graphical screens (able to display a wide range of real-time and historical data).

An IPD may also include an interactive interface, such as a keypad or touch-screen, to allow user input.

Programmable Communicating Thermostat (PCT)

The Programmable Communicating Thermostat (PCT) provides a means of controlling a heating and/or air-conditioning system, allowing intervention from the utility company through the demand-response feature introduced in Section 1.1.2. For example, during periods of high energy demand, the utility company may send a request to the PCT to adjust the system’s temperature setting or to shut the system down completely. However, a user over-ride feature would normally be provided. The PCT behaves in a similar way to the Load Control Device (described below). A PCT is likely to feature a user interface (buttons and a display), and it may be desirable to incorporate the PCT and IPD in the same physical unit.

Load Control Device

The Load Control Device provides a means of controlling a high-power appliance (e.g. water heater, swimming pool pump), allowing intervention from the utility company through the demand-response feature introduced in Section 1.1.2. For example, during periods of high energy demand, the utility company may send a request to the Load Control Device to reduce the appliance’s power setting or to shut the appliance down completely. However, a user over-ride feature would normally be provided.

Smart Appliance

The Smart Appliance is a device incorporated in a commercial product (e.g. washing machine) to allow the product to take action on receiving communications from the utility company. The action may be to simply display a message (such as a warning of a high-price period) or to implement a decision (for example, to automatically delay the operation of a washing machine or to reduce its temperature setting during a high-price period).

Prepayment Terminal

The Prepayment Terminal is used only by customers who pay for their power in advance in discrete amounts. The device allows a payment to be made (e.g. using coins or a bank/credit card) and also displays messages (such as the remaining prepayment balance and warnings when the balance approaches zero).

1.3 Smart Energy Networks

The Smart Energy devices on a customer's premises form a Home Area Network (HAN) which is installed and owned by the utility company, and therefore referred to as the "utility private HAN". This local network is connected via the Energy Service Portal (ESP) to the utility company's backhaul network, over which messages can be exchanged between the HAN and the company's headquarters.

In this manual, we are concerned with HANs that are implemented as wireless networks which employ the ZigBee PRO protocol operating in the 2.4-GHz radio band. This wireless solution minimises installation costs and effort, and also causes minimal disruption at customer premises. Smart Energy networks from the ZigBee perspective are described in more detail in Chapter 2.

Smart Energy HANs and related network issues are described in the sub-sections below.

1.3.1 Home Area Networks (HANs)

This section describes Smart Energy HANs by starting with the simplest HAN and building up to more complex HANs.

SE Network with Automated Meter Reading (only)

The simplest HAN contains an ESP and a Metering Device, as illustrated in Figure 1 below. This type of HAN facilitates Automated Meter Reading (AMR) only.

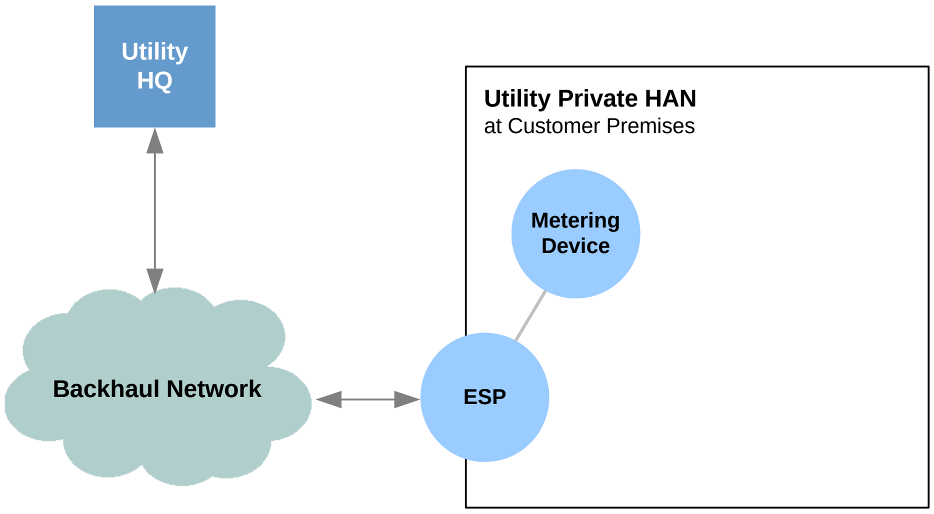

Figure 1: Simplest Utility Private HAN

In practice, the ESP and Metering Device are often incorporated in a single device, as shown in Figure 3.

SE Network with Display Capabilities

Adding an In-Premise Display (IPD) to the above HAN (in Figure 1) provides the consumer with useful information which enables them to manage their power consumption in real-time and realise energy/cost savings. This basic but effective Smart Energy HAN is illustrated in Figure 2 below.

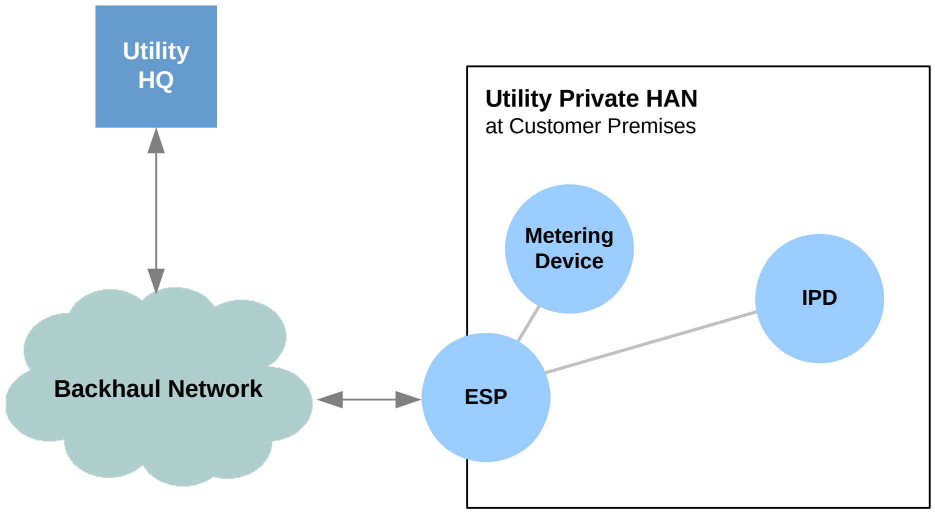

Figure 2: Utility Private HAN with IPD

The figure below shows the same utility private HAN with the ESP and Metering Device combined into a single device. However, in the remainder of this chapter, they are shown as separate devices.

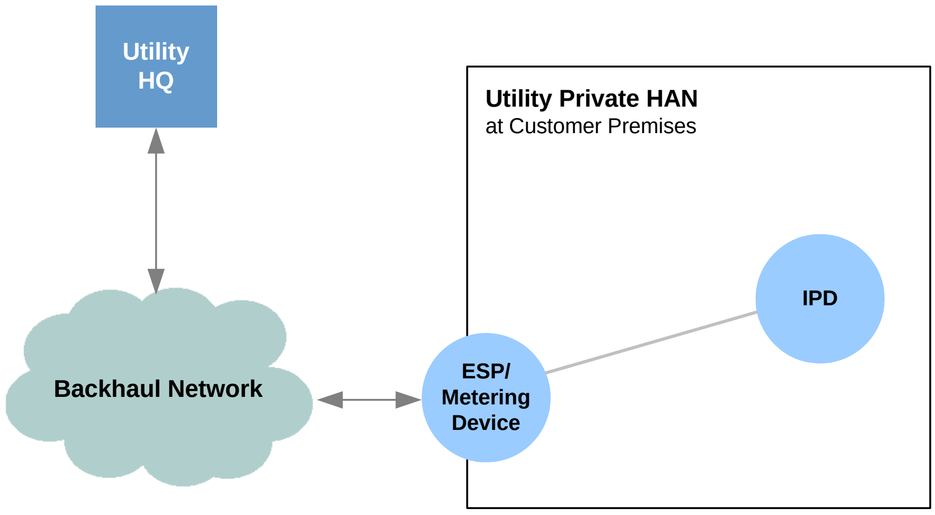

Figure 3: Utility Private HAN with IPD and Combined ESP/Metering Device

SE Network with Automated Energy Management

Further Smart Energy devices (from those listed in Section 1.2) can be added to the above HAN (in Figure 2) to achieve automated energy management that does not rely on customer decisions and actions. Such a system is illustrated in Figure 4 below.

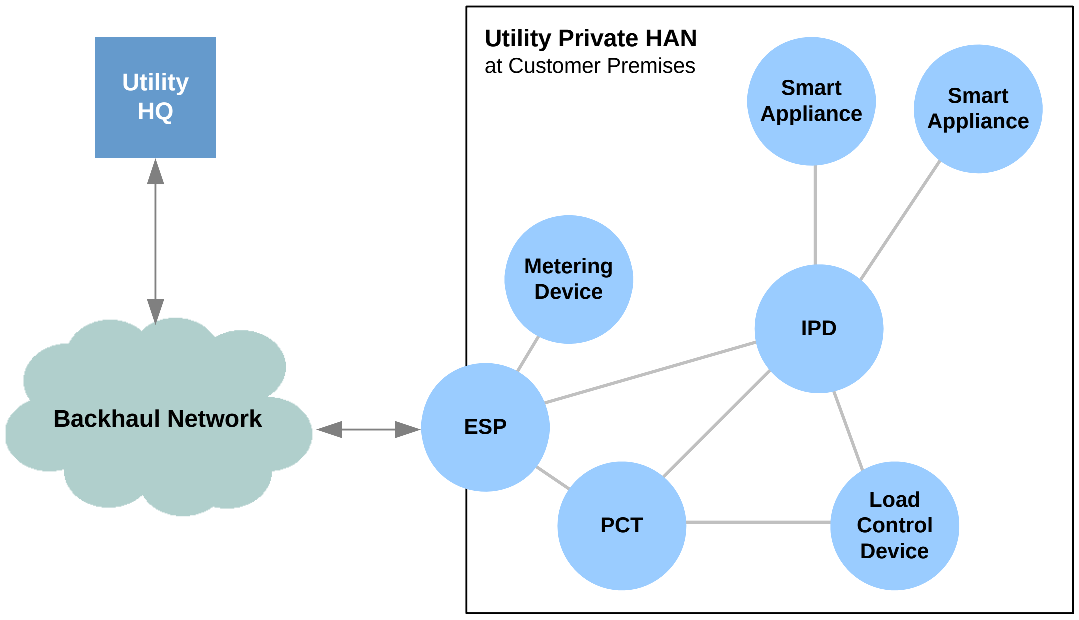

Figure 4: Utility Private HAN with Range of SE Devices

SE Network with Utility and Customer Sectors

It is also possible for a customer to have their own private HAN which is connected to the utility private HAN. In this case, there must be a bridge device between the two HANs. In the customer private HAN, only the bridge device needs to be registered with the ESP of the utility private HAN (other devices in the customer private HAN do not). The customer private HAN could be installed by the customer or by a home automation professional. Such a system is illustrated in Figure 5 below.

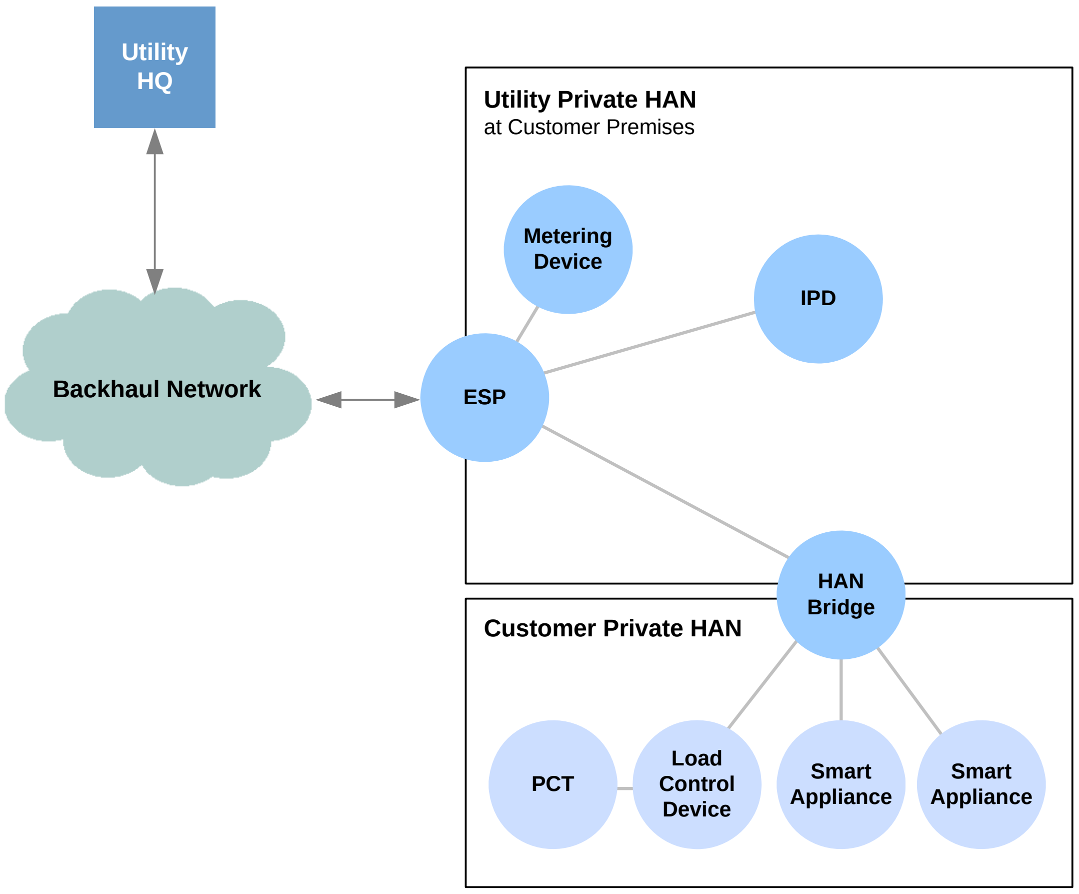

Figure 5: Utility Private HAN with Customer Private HAN

Note that the customer private HAN may have its own connection to the outside world via the internet (not shown in the above diagram). This provides an alternative route for obtaining information from the utility company.

1.3.2 Neighbourhood Area Networks (NANs)

In some situations, the ESP of the utility private HAN may not connect directly to the backhaul network, but via an intermediate Neighbourhood Area Network (NAN) which also belongs to the utility company - the “utility private NAN”. This may be the case for apartments in a block, in which each apartment has a utility private HAN that connects via its ESP to the utility private NAN for the whole block. The NAN has an ESP which connects it to the backhaul network. This is illustrated in Figure 6 below.

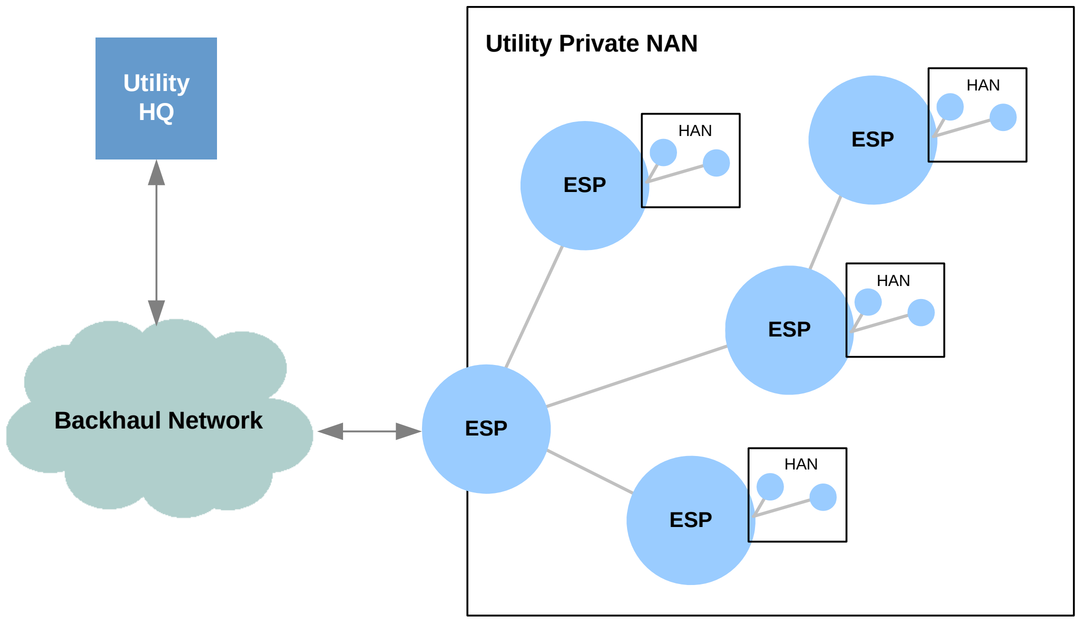

Figure 6: Utility Private NAN

1.3.3 Network Security

Security is a major concern in a Smart Energy network due to the need to ensure that a utility private HAN:

- cannot be ‘hacked into’ for the purpose of corrupting power consumption data
- cannot be accessed by a neighbouring HAN

The ZigBee PRO wireless network protocol, on which this manual is based, provides security measures based on digital certificates and keys used in exchanging messages within the network. In addition, as standard, ZigBee PRO employs network identifiers and intelligent radio channel selection to prevent neighbouring networks from accidentally interfering with each other. Further information on the application of ZigBee PRO to Smart Energy networks is provided in Chapter 2, with network security described in Section 2.5.

2. ZigBee Smart Energy

This chapter provides the essential information for an understanding of the implementation of a Smart Energy (SE) network using the ZigBee PRO wireless network protocol. It assumes that you are already familiar with the basic Smart Energy concepts presented in Chapter 1.

2.1 Essential ZigBee Concepts

In this manual, we are concerned with implementing a Smart Energy HAN as a ZigBee PRO wireless network. ZigBee PRO is a second generation wireless network protocol, defined by the ZigBee Alliance, which provides a worldwide standard for the implementation of highly flexible 'mesh' networks. The protocol is built on top of the IEEE 802.15.4 standard. The NXP implementation of ZigBee PRO operates in the 2.4-GHz radio band, which allows licence-free radio operation in most parts of the world (check your local regulations).

A full introduction to ZigBee PRO is provided on the *ZigBee PRO Stack User Guide (JN-UG-3048)*, available from **www.nxp.com/jennic**. The rest of this section describes specific ZigBee concepts relevant to Smart Energy, before the ZigBee PRO Smart Energy profile is introduced in the remainder of this chapter.

While reading through this section, you may wish to refer to the representation of the ZigBee PRO protocol stack in Figure 7 below.

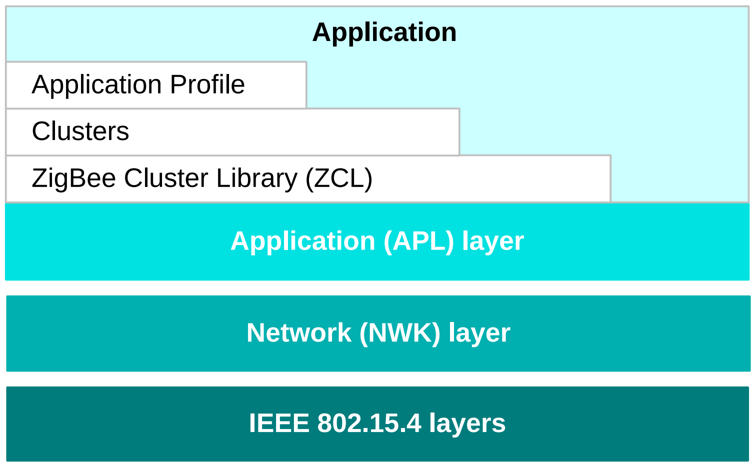

Figure 7: ZigBee PRO Stack

2.1.1 Application Profiles

ZigBee is designed as an industry-standard protocol for the implementation of wireless networks by different manufacturers. One of its aims is to allow wireless network devices from multiple manufacturers to operate together in the same network. With this aim in mind, the ZigBee Alliance introduced the concept of an Application Profile.

An Application Profile provides a design framework for a specific market sector by defining a set of devices that can be coherently used together in implementing an application for that market sector. For example, the ZigBee Alliance has defined the Home Automation (HA) profile for use in controlling appliances and systems in the home, such as a lighting system. It defines a number of 'devices' and functions that are needed or useful in controlling domestic systems, e.g. switches, dimmers, occupancy sensors and load controllers for a lighting system.

Note: In ZigBee, a 'device' is a software entity which encompasses a particular set of properties and functionality. This is explained below in Section 2.1.2.

The ZigBee Alliance has defined a public Application Profile for Smart Energy. This is introduced in Section 2.2 and forms the basis for the rest of this manual.

2.1.2 Devices, Clusters and Attributes

A ZigBee Application Profile, such as Smart Energy, incorporates the set of devices that it supports. In ZigBee, a 'device' is a software entity comprising the set of properties and functionality of an application that runs on a network node - this information is detailed in its device descriptors (Node, Power and Simple descriptors).

Note: An application which runs on a network node has an associated endpoint, which is effectively the application's input/output port. Up to 240 endpoints are available for use by the applications on each node, and are numbered 1 to 240.

The Application Profile also defines the type of data supported by the application and the operations that can be performed on the data. These definitions are handled in terms of 'attributes' and 'clusters', as described below.

Attributes

An attribute is a data entity, such as temperature. An application has a set of attributes that it uses in its communications - attribute values are exchanged in the messages sent from one application to another in a ZigBee network (see Clusters below).

Clusters

ZigBee applications use the concept of a ‘cluster’ for communicating attribute values. A cluster comprises a set of related attributes together with a set of commands.

A cluster has two aspects, which are respectively concerned with receiving and sending commands. One or both aspects may be used by a ZigBee application device. These sides of a cluster are described below and illustrated in Figure 8.

- **Server or Input Cluster:** This side of a cluster is used to store attributes and receive commands to manipulate the stored attributes (to which the cluster may return responses) - for example, a server cluster on a Metering Device may store an “energy consumed” attribute and associated attributes, and respond to commands which request readings of these attributes.
- **Client or Output Cluster:** This side of a cluster is used to manipulate attributes in the corresponding server cluster by sending commands to it (and receiving the responses). Normally, these are read commands to obtain attribute values (the read values being returned in the responses) - for example, to read the “energy consumed” attribute (and associated attributes) on a remote Metering Device. There may also be write commands to remotely set attribute values.

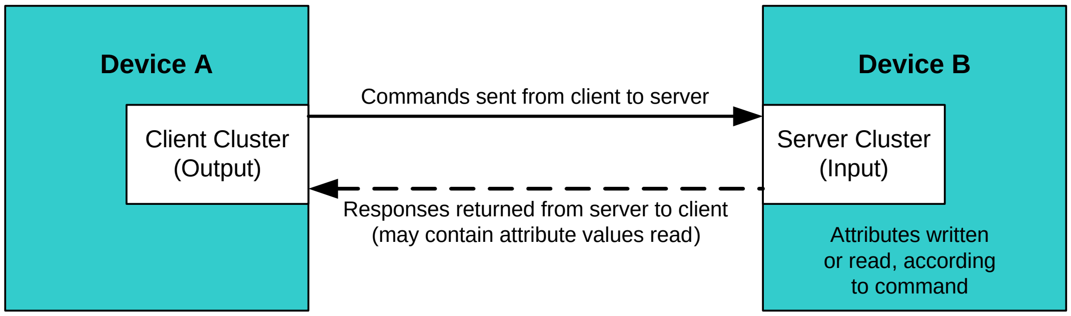
Device A
Client Cluster (Output)
Commands sent from client to server
Responses returned from server to client (may contain attribute values read)
Device B
Server Cluster (Input)
Attributes written or read, according to command

Figure 8: Client and Server Clusters

The input clusters and output clusters of an application are listed (separately) in its Simple descriptor, which forms part of the Application Profile.

2.1.3 ZigBee Cluster Library (ZCL)

Although clusters may be defined in an Application Profile, certain clusters are useful across all Application Profiles. Therefore, the ZigBee Alliance has defined a number of standard clusters for different functional areas. These are collected together in the ZigBee Cluster Library (ZCL). For example, the Smart Energy profile uses the Time cluster from the ZCL (for the synchronisation of the nodes in an SE network).

The ZCL provides a common means for applications to communicate. It defines a header and payload that sit inside the Protocol Data Unit (PDU) used for messages. It also defines attribute types (such as ints, strings, etc), common commands (e.g. for reading attributes) and default responses for indicating success or failure.

The ZCL is fully detailed in the *ZigBee Cluster Library Specification (075123)*, available from the ZigBee Alliance.

2.2 ZigBee SE Profile

The ZigBee Alliance has published a public Application Profile for Smart Energy, public in the sense that it is available to all manufacturers who wish to develop Smart Energy products for ZigBee networks. This Application Profile ID is numbered 0x0109 and the profile is defined in the *ZigBee Smart Energy Profile Specification (075356)*, available from the ZigBee Alliance. However, this User Guide should provide the necessary information to use the profile.

Some general points to note about the implementation of the ZigBee SE profile are as follows:

- In scanning for channels, the ZigBee Co-ordinator should ideally restrict itself to the following 2.4-GHz band channels: 11, 14, 15, 19, 20, 24, 25. This is to avoid frequency overlap with nearby Wi-Fi systems.
- Broadcasts are strongly discouraged in SE networks and are limited to no more than one broadcast per second (and much less frequently being preferable).
- ZigBee PRO SE networks are required to use the protocol's standard security mode with the obligatory use of link keys.

The SE profile specifies a set of clusters, listed in Section 2.3, as well as a set of possible SE devices that use these clusters, detailed in Section 2.4. Security is an important factor in SE networks and the SE profile's security requirements are outlined in Section 2.5. The SE profile also defines required aspects of network operation, including start-up attribute sets (containing global network variables such as PAN ID and channel mask).

2.3 ZigBee SE Clusters

The ZigBee SE profile uses certain clusters from the ZCL and also defines some of its own clusters. All the clusters used by the SE profile are listed in Table 1 and outlined below. In the table, the clusters from the ZCL are listed as “General” and the clusters defined by the SE profile are listed as “Smart Energy”.

Category	Cluster	Cluster ID
General	Basic	0x0000
	Power Configuration	0x0001
	Identify	0x0003
	Alarms	0x0009
	Time	0x000A
	Commissioning	0x0015
	Over-the-Air (OTA) Upgrade	0x0019
Smart Energy	Price	0x0700
	Demand-Response and Load Control	0x0701
	Simple Metering	0x0702
	Messaging	0x0703
	Tunnelling (Complex Metering)	0x0704
	Prepayment	0x0705
	Key Establishment	0x0800

Table 1: SE Profile Clusters

Note 1: The Smart Energy clusters are detailed in this manual. The General/ZCL clusters are detailed in the *ZCL User Guide (JN-UG-3077)* - only SE-specific implementation details of the ZCL clusters are described in this manual.

Note 2: The above table lists the clusters included in the ZigBee Smart Energy Profile Specification, but not all clusters are supported in the NXP ZigBee PRO Smart Energy API. A list of the supported clusters (and attributes) is provided in Appendix A.

Basic

The Basic cluster contains the basic properties of a device (e.g. software and hardware versions) and allows the setting of user-defined properties (such as location). The Basic cluster is described further in Chapter 5.

Power Configuration

The Power Configuration cluster contains the properties of the device's own power source (which can be mains or batteries), and allows the configuration of under/over voltage alarms. The Power Configuration cluster is described further in the *ZCL User Guide (JN-UG-3077)*.

Identify

The Identify cluster allows a device to make itself known visually (e.g. by flashing a light) to an observer such as a network installer.

Alarms

The Alarms cluster is used for sending alarm notifications and the general configuration of alarms for all other clusters on the device (individual alarm conditions are set in the corresponding clusters).

Time

The Time cluster provides an interface to a real-time clock on a device, allowing the clock time to be read and written in order to synchronise the clock to a time standard - the number of seconds since 0 hrs 0 mins 0 secs on 1st January 2000 UTC (Co-ordinated Universal Time). This cluster includes functionality for local time-zone and daylight saving time. The Time cluster is described further in Chapter 5.

Commissioning

The Commissioning cluster can be optionally used for commissioning the ZigBee stack on a device (during network installation) and defining the device behaviour with respect to the ZigBee network (it does not affect applications operating on the devices). The Commissioning cluster is described further in the *ZCL User Guide (JN-UG-3077)*.

OTA Upgrade

The Over-the-Air (OTA) Upgrade cluster provides the facility to upgrade application software on the nodes of a ZigBee PRO network by distributing the replacement software through the network (over the air) from a designated node. In an SE network, this node is normally the ESP. The OTA Upgrade cluster is described further in Chapter 5.

Price

The Price cluster provides the mechanism for sending and receiving pricing information within an SE network. The pricing information is sent by the utility company to the ESP, which passes the information to SE devices in the utility private HAN (and possibly to other devices in a customer private HAN). The Price cluster is described further in Chapter 6.

Demand-Response and Load Control

The Demand-Response and Load Control (DRLC) cluster provides an interface for controlling an attached appliance that supports load control. The cluster is able to receive load control requests (from the utility company) and act upon them - the demand-response functionality. The DRLC cluster is described further in Chapter 9.

Simple Metering

The Simple Metering cluster provides a mechanism to obtain consumption data from a metering device (electric, gas, water or thermal). The Simple Metering cluster is described further in Chapter 8.

Messaging

The Messaging cluster provides an interface for passing text messages between ZigBee SE devices. These are likely to be messages from the utility company received by the ESP, which unicasts a received message to all registered SE devices that implement the Messaging cluster (or makes the message available to all devices for later collection). The Messaging cluster is described further in Chapter 7.

Tunnelling (Complex Metering)

The Tunnelling cluster provides an interface for implementing tunnelling protocols. It allows any existing metering communication protocol to be transported within the payload of standard ZigBee frames (this includes dealing with issues such as addressing, fragmentation and flow control). The Tunnelling cluster is described further in Chapter 11.

Prepayment

The Prepayment cluster provides support for pre-purchased credit which, in particular, can be used for electricity consumption. The cluster provides the facility to exchange prepayment information between devices in a HAN - for example, to allow prepayment data held on a Metering Device to be displayed to the customer on an In-Premise Display (IPD).

Key Establishment

The Key Establishment cluster is used to manage secure communications in ZigBee, when the underlying network security cannot be trusted. A key agreement scheme is used, requiring the exchange of keying information between communicating devices. Security in SE networks is described further in Section 2.5. The Key Establishment cluster is described further in Chapter 10.

2.4 ZigBee SE Devices

The ZigBee SE profile defines the following devices for use in SE networks:

- Energy Service Portal (ESP)
- Metering Device
- In-Premise Display (IPD)
- Programmable Communicating Thermostat (PCT)
- Load Control Device
- Smart Appliance
- Range Extender

The general roles of these devices are as described in Section 1.2, except the Range Extender which is simply a ZigBee Router. Also note that the Prepayment Terminal, mentioned in Section 1.2, is not yet supported by ZigBee.

The SE devices are defined by the clusters that they use. Some clusters are common to all the SE devices - these are detailed in Table 2 below.

Note: For each device, there are mandatory clusters and optional clusters. Also, the clusters are different for the server (input) and client (output) sides of the device.

Server (Input) Side	Client (Output) Side
Mandatory	
Basic	
Key Establishment	Key Establishment
Optional	
Identity	
Clusters with reporting capability	Clusters with reporting capability
Power Configuration	
Inter-PAN Communication	Inter-PAN Communication
Alarms	
Commissioning	Commissioning
OTA Upgrade	OTA Upgrade
Tunnelling	Tunnelling
Manufacturer-specific	Manufacturer-specific

Table 2: Common Clusters for All SE Devices

The SE devices are further described in the sub-sections below, which include details of the additional clusters that each device uses.

2.4.1 Energy Service Portal (ESP)

The Energy Service Portal (ESP) acts as the exit/entry point for the utility private HAN, connecting the HAN to the utility company’s backhaul network and routing messages into and out of the HAN. The ESP normally acts as the ZigBee Co-ordinator and also includes the appropriate non-ZigBee interface to communicate on the backhaul network. Every SE network must have an ESP, which may be housed with another SE device, often a Metering Device, with both devices operating on the same ZigBee endpoint.

The specific clusters used by the ESP (in addition to the common clusters in Table 2) are listed in Table 3 below.

Server (Input) Side	Client (Output) Side
Mandatory	
Messaging	
Price	
Demand-Response/Load Control	
Time	
Optional	
	Price
Simple Metering	Simple Metering

Table 3: Additional Clusters for ESP Device

2.4.2 Metering Device

A Metering Device measures the consumption of a resource (normally electricity, gas, water or heat). The device may be able to accept and respond to read requests, or may automatically report its measurements periodically. The device may also communicate certain status indicators (e.g. battery low, tamper detected). A Metering Device for electricity consumption is often housed with the ESP, with both devices operating on the same ZigBee endpoint - this is the case in the NXP ZigBee PRO SE implementation.

The specific clusters used by the Metering Device (in addition to the common clusters in Table 2) are listed in Table 4 below.

Server (Input) Side	Client (Output) Side
Mandatory	
Simple Metering	
Optional	
	Time
	Price
	Messaging

Table 4: Additional Clusters for Metering Device

2.4.3 In-Premise Display (IPD)

The In-Premise Display (IPD) device relays consumption information to the user by some kind of visual means (e.g. LEDs, text display, graphical display). The device typically displays at least one of:

- current power usage
- consumption history over selectable periods
- pricing information
- text messages

For example, the device may be used to display warnings of high-price periods so that the consumer can decide whether to modify their power usage in real-time.

The IPD may be fully interactive with some kind of user input interface (e.g. buttons, keypad, touch screen).

The specific clusters used by the IPD (in addition to the common clusters in Table 2) are listed in Table 5 below.

Server (Input) Side	Client (Output) Side
Optional	
	Demand-Response/Load Control
	Time
	Price
	Simple Metering
	Messaging

Table 5: Additional Clusters for IPD Device

Since there are no mandatory clusters for the IPD, the device must implement at least one of the optional clusters. NXP have implemented the Basic client-side cluster to allow the IPD to read the Basic cluster attributes on the ESP.

2.4.4 Programmable Communicating Thermostat (PCT)

The Programmable Communicating Thermostat (PCT) is used to manage the power consumption of a heating and/or air-conditioning system. The device supports the demand-response feature, meaning that it is able to receive requests from the utility company for changes in power consumption (e.g. to reduce consumption during high-demand periods) and act on these requests.

The specific clusters used by the PCT (in addition to the common clusters in Table 2) are listed in Table 6 below.

Server (Input) Side	Client (Output) Side
Mandatory	
	Demand-Response/Load Control
	Time
Optional	
	Price
	Simple Metering
	Messaging

Table 6: Additional Clusters for PCT Device

2.4.5 Load Control Device

The Load Control device is used to manage the power consumption of an attached appliance, usually a high-power appliance such as a water heater or swimming pool pump. The device supports the demand-response feature, meaning that it is able to receive requests from the utility company for changes in power consumption (e.g. to reduce consumption during high-demand periods) and act on these requests.

The specific clusters used by the Load Control device (in addition to the common clusters in Table 2) are listed in Table 7 below.

Server (Input) Side	Client (Output) Side
Mandatory	
	Demand-Response/Load Control
	Time
Optional	
	Price

Table 7: Additional Clusters for Load Control Device

2.4.6 Smart Appliance

The Smart Appliance is a device that can be built into a consumer product, such as a washing machine, to allow it to participate directly in a ZigBee SE network. The device is able to receive information (at least pricing data) from the utility company and act on it (e.g. display information to the consumer).

The specific clusters used by the Smart Appliance device (in addition to the common clusters in Table 2) are listed in Table 8 below.

Server (Input) Side	Client (Output) Side
Mandatory	
	Price
	Time
Optional	
	Demand-Response/Load Control
	Messaging

Table 8: Additional Clusters for Smart Appliance Device

2.4.7 Range Extender

The Range Extender is a ZigBee Router used for relaying messages between the nodes of an SE network. A product which implements the Range Extender device cannot implement any other device from the SE profile, although it is possible to implement a private application alongside the Range Extender device.

Only the common clusters (listed in Table 2) are used by the Range Extender device.

2.5 ZigBee SE Security

All communications in an SE network are secured to protect the network from intentional and unintentional interference. To this end, ZigBee PRO incorporates a number of security features. In addition, the SE profile provides security enhancements concerned with establishing the security keys used in network communications. These are outlined in the sub-sections below.

2.5.1 ZigBee PRO Security

ZigBee PRO has default features that isolate a network from neighbouring networks, ensuring that there is no cross-over between networks. During network start-up, the ZigBee Co-ordinator (normally incorporated in the ESP of an SE network) performs the following steps:

1. Sets the 64-bit Extended PAN ID (EPID) for the network, which can be taken from the IEEE/MAC address of the Co-ordinator - this is a globally unique 64-bit address and so the resulting EPID will uniquely identify the network.
2. Scans the permissible radio channels in order to select the network's operating channel, which is normally chosen to be the channel with the least detected activity, thus avoiding clashes with other networks.
3. Randomly selects a 16-bit PAN ID for the network, ensuring that the chosen value does not clash with the PAN ID of any other ZigBee network operating in the same channel.

The above measures should protect a ZigBee network from accidental interference from other ZigBee networks in the neighbourhood.

ZigBee PRO also provides specific security features which help protect the network from malicious attacks. These are described in the *ZigBee PRO Stack User Guide (JN-UG-3048)*. The ZigBee PRO 'standard security' mode is used in SE networks.

ZigBee PRO security employs AES-CCM* encryption. This is a very high-security, 128-bit key-based encryption system which is applied to network communications, preventing external agents from interpreting ZigBee network data.

Two types of security key are used in ZigBee PRO encryption:

- **Network key:** This key is randomly generated by the Trust Centre and is shared by all network nodes. It is used to secure all communications between nodes (even when an application link key is also used - see below).
- **Application link key:** This is a unique link key for each pair of communicating nodes. The application link key provides a high level of security that is used in accessing certain SE clusters (e.g. Simple Metering, Price, Messaging).

Use of the above keys in a Smart Energy network is described further in the next section.

2.5.2 Smart Energy Security

ZigBee PRO SE devices are required to use application link keys (see Section 2.5.1) for encrypted communication between pairs of nodes. The SE profile provides a Key Establishment cluster (which is mandatory for all devices) to establish unique application link keys, as outlined below.

In order to establish an application link key between the Co-ordinator/ESP and a joining node (any SE device), the following are required:

- Pre-configured link key for the joining node
- Security certificates for the joining node and the Trust Centre

Information on how to obtain the above key and certificates is given below in the sections Pre-configured Link Key and Security Certificate respectively. The procedure below describes the generation of the application link key, which is also illustrated in Figure 9. In this procedure, the above key and relevant certificate are assumed to be already held by the joining node and by the Trust Centre, which is assumed to be the Co-ordinator/ESP.

1. The joining node sends a 'join request' to the Co-ordinator/Trust Centre, which returns a 'transport key' containing the network key encrypted using the pre-configured link key of the newly joined node.
2. The Key Establishment cluster on the joining node then uses both nodes' security certificates to generate an application link key through a sequence of exchanges with the Co-ordinator, encrypted using the network key.
3. The established application link key can subsequently be used to encrypt communications between the joined node and the Co-ordinator.

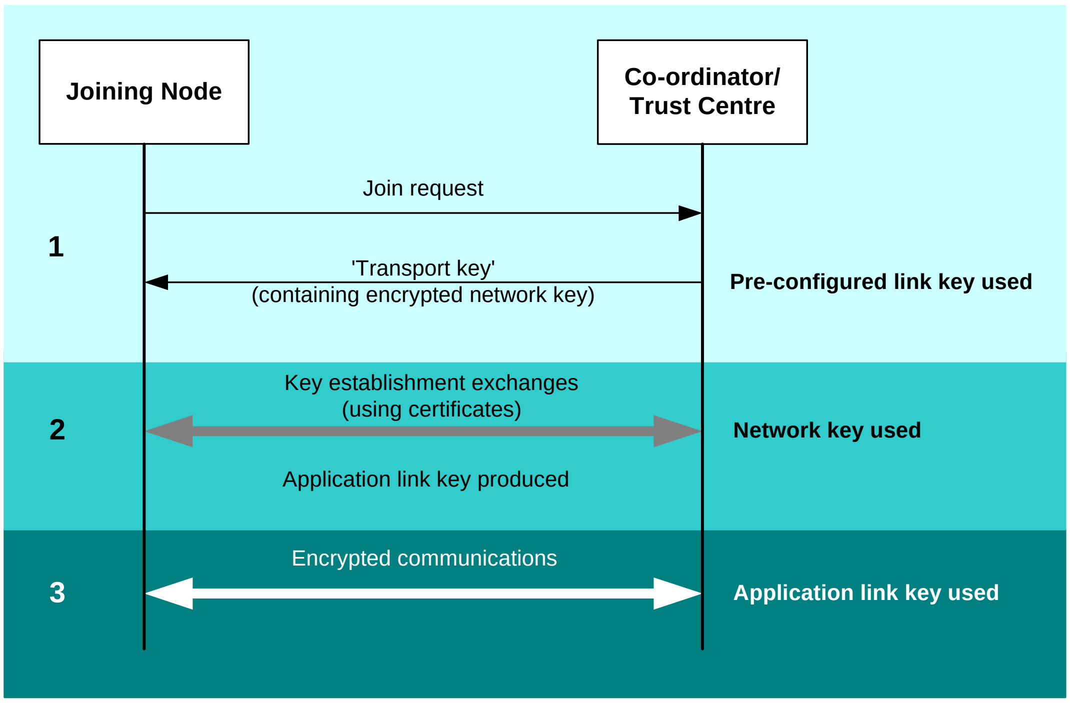
Joining Node
Co-ordinator/ Trust Centre
Join request
1
'Transport key' (containing encrypted network key)
Pre-configured link key used
Key establishment exchanges (using certificates)
2
Network key used
Application link key produced
Encrypted communications
3
Application link key used

Figure 9: Application Link Key Establishment

For further details of the Smart Energy security measures and the Key Establishment cluster, refer to the *ZigBee Smart Energy Profile Specification (075356)*.

The pre-configured link key and security certificates are obtained as described below.

Pre-configured Link Key

As mentioned above, SE security set-up requires a pre-configured link key for the joining node. This link key is shared between the joining node and the Trust Centre/ ESP as described below.

1. During node manufacture, the node is assigned an installation code, which is printed on a label distributed with the node. This installation code consists of 12, 16, 24 or 32 random hex digits, followed by a 4-digit checksum of the random digits. The leading random (non-checksum) digits are also used in an algorithm to derive the node's 128-bit pre-configured link key, which is pre-programmed in the Flash memory of the device during manufacture.
2. During node installation in the network, the node's IEEE/MAC address and installation code are communicated to the utility company via an out-of-band mechanism (e.g. telephone call or web site registration), with the embedded checksum being used to validate the registered installation code. The utility company then derives the pre-configured link key from the leading random (non-checksum) digits of this code, and installs this key and the MAC address into the ESP/Trust Centre of the SE network via the backhaul network. The new node is then powered on and attempts to join the network (see above).

Security Certificate

SE security set-up also requires digital security certificates for the joining node and the Trust Centre. This certificate contains the following information: IEEE/MAC address of the node, issuer, profile attribute data, a public key and the signature of a Certificate Authority (CA). The certificate also has an associated but separate private key (for information on the use of public and private keys, refer to http://www.verisign.com.au/ repository/tutorial/cryptography/intro1.shtml).

A security certificate for a device can be obtained from Certicom (www.certicom.com) by submitting the IEEE/MAC address of the device. The "issuer" embedded in the certificate is then an identifier for Certicom (a unique IEEE/MAC address for the company). Note that Certicom issue test certificates and production certificates. Production certificates are issued under a higher level of security and to obtain a production certificate, your device must already have ZigBee Smart Energy certification.

3. Smart Energy Application Development

This chapter provides basic guidance on developing a ZigBee PRO SE application. The topics covered in this chapter include:

- Development resources and their installation (Section 3.1)
- Smart Energy Application Programming Interface (API) (Section 3.2)
- API functions (Section 3.3)
- Development phases (Section 3.4)
- Building an application (Section 3.5)

Application coding is described separately in Chapter 4.

3.1 Development Resources and Installation

NXP provide a wide range of resources to aid in the development of Smart Energy applications for the JN51xx wireless microcontroller. An SE application is developed as a ZigBee PRO application that uses the NXP ZigBee PRO APIs in conjunction with JenOS (Jennic Operating System), together with SE-specific and ZCL resources. All resources are available from **www.nxp.com/jennic** and are outlined below.

ZigBee PRO Resources

The resources for developing a ZigBee PRO application are supplied free-of-charge in a Software Developer's Kit (SDK), which is provided as two installers:

- **JN516x ZigBee PRO Smart Energy SDK (JN-SW-4064):** This installer contains a number of APIs, including the ZigBee PRO APIs, JenOS APIs and Integrated Peripherals API for the JN516x device.
- **SDK Toolchain (JN-SW-4041):** This installer contains the tools that you will use in creating an application, including the Eclipse IDE (Integrated Development Environment) and the JN51xx Flash Programmer.

For full details of the SDK and installation instructions, refer to the *SDK Installation and User Guide (JN-UG-3064)*. The SDK is normally installed into the directory **C:/Jennic**.

SE Resources

The SE resources comprise of software files relating to the clusters and devices of the ZigBee Alliance's SE profile, as well as the relevant clusters of the ZCL. These resources are included as source in the SE SDK installer (JN-SW-4064) described above.

A Smart Energy demonstration application is provided in the Application Note *Smart Energy HAN Solutions (JN-AN-1135)*, available from **www.nxp.com/jennic**.

3.2 Smart Energy API

The NXP ZigBee PRO Smart Energy API contains a range of resources (functions, structures, etc), including:

- Core resources (e.g. for initialising the API and accessing SE cluster attributes)
- Cluster-specific resources

These resources are introduced in the sub-sections below.

3.2.1 Core Resources

The core resources of the Smart Energy API handle the basic operations required in an SE network, irrespective of the devices and clusters used:

- Initialising the SE API
- Registering an endpoint on a SE device
- Requesting a read access to cluster attributes on a remote device
- Requesting a write access to cluster attributes on a remote device
- Handling errors from SE API function calls
- Handling events on an SE device

The core resources and their use are described in Chapter 4, Chapter 12 and Chapter 13.

3.2.2 Cluster-specific Resources

A ZigBee PRO Smart Energy device uses certain mandatory and optional ZigBee clusters (for details, refer to Chapter 2). The clusters supported by the NXP Smart Energy software are listed below:

- Basic cluster (from ZCL) - see Chapter 5
- Time cluster (from ZCL) - see Chapter 5
- Identify cluster (from ZCL) - see Chapter 5
- Commissioning cluster (from ZCL) - see Chapter 5
- OTA Upgrade cluster (from ZCL) - see Chapter 5
- Price cluster - see Chapter 6
- Messaging cluster - see Chapter 7
- Simple Metering cluster - see Chapter 8
- Demand-Response and Load Control cluster - see Chapter 9
- Key Establishment cluster - see Chapter 10

Other SE clusters from the ZigBee SE Profile and ZCL are not yet supported by the NXP SE software.

3.3 Function Prefixes

The API functions used in SE are categorised and prefixed in the following ways:

- **ZCL functions:** Used to interact with the ZCL and prefixed with **xZCL_**
- **SE functions:** Used to interact with the SE profile and prefixed with **xSE_**
- **Cluster functions:** Used to interact with clusters and prefixed as follows:
  - For clusters defined in the SE specification, they are prefixed with **xSE_**
  - For clusters defined in the ZCL specification, they are prefixed with **xCLD_**
  - For the OTA Upgrade cluster, they are prefixed with **xOTA_**

In the above prefixes, x represents one or more characters that indicate the return type, e.g. “v” for **void**.

Only functions that are SE-specific are detailed in this manual. Functions which relate to clusters of the ZCL are detailed in the *ZCL User Guide (JN-UG-3077)*.

3.4 Development Phases

The main phases of development for an SE application are the same as for any ZigBee PRO application, and are outlined below.

Note: Before starting your SE application development, you should familiarise yourself with the general aspects of ZigBee PRO application development, described in the *ZigBee PRO Stack User Guide (JN-UG-3048)*.

1. **Network Configuration:** Configure the ZigBee network parameters for the nodes using the ZPS Configuration Editor - refer to the *ZigBee PRO Stack User Guide (JN-UG-3048)* and to Section 13.5 of this manual.
2. **OS Configuration:** Configure the JenOS resources to be used by your application using the JenOS Configuration Editor - refer to the *JenOS User Guide (JN-UG-3075)*.
3. **Application Code Development:** Develop the application code for your nodes using the ZigBee PRO APIs, JenOS APIs, SE API and ZCL - refer to the *ZigBee PRO Stack User Guide (JN-UG-3048), JenOS User Guide (JN-UG-3075)* and *ZCL User Guide (JN-UG-3077),* as well as this manual.
4. **Application Build:** Build the application binaries for your nodes using the JN51xx compiler and linker built into the Eclipse platform - refer to Section 3.5 and to the *SDK Installation and User Guide (JN-UG-3064)*.
5. **Node Programming:** Load the application binaries into Flash memory on your nodes using the JN51xx Flash programmer, which can be launched either from within Eclipse or directly, and is described in the *JN51xx Flash Programmer User Guide (JN-UG-3007)*.

3.5 Building an Application

This section outlines how to build a ZigBee PRO Smart Energy application developed for the JN51xx device. First of all, the configuration of compile-time options and ZigBee Device Parameters is described, and then directions are given for building and loading the application.

3.5.1 Compile-Time Options

Before the application can be built, the SE compile-time options must be configured in the header file **zcl_options.h** for the application. This header file is supplied in the Application Note *Smart Energy HAN Solutions (JN-AN-1135)*, which can be used as a template.

Number of Endpoints

The highest numbered endpoint used by the SE application must be specified - for example:

```
#define SE_NUMBER_OF_ENDPOINTS   3
```

Normally, the endpoints starting at endpoint 1 will be used for SE, so in the above case endpoints 1 to 3 will be used for SE. It is possible, however, to use the lower numbered endpoints for non-SE purposes - for example, to run other protocols on endpoints 1 and 2, and SE on endpoint 3. In this case, with SE_NUMBER_OF_ENDPOINTS set to 3, some storage will be statically allocated by the SE library for endpoints 1 and 2 but never used. Note that this define applies only to local endpoints - the application can refer to remote endpoints with numbers beyond the locally defined value of SE_NUMBER_OF_ENDPOINTS.

Enabled Clusters

All required clusters must be enabled in the options header file. For example, a minimal meter application must have the Basic and Simple Metering clusters:

```
#define CLD_BASIC
#define CLD_SIMPLE_METERING
```

Support for Attribute Read/Write

Read/write access to cluster attributes must be explicitly compiled into the application, and must be enabled separately for the server and client sides of a cluster using the following macros in the options header file:

```
#define ZCL_ATTRIBUTE_READ_SERVER_SUPPORTED
#define ZCL_ATTRIBUTE_READ_CLIENT_SUPPORTED
#define ZCL_ATTRIBUTE_WRITE_SERVER_SUPPORTED
#define ZCL_ATTRIBUTE_WRITE_CLIENT_SUPPORTED
```

Note that each of the above definitions will apply to all clusters used in the application.

In the current NXP implementation of ZigBee PRO Smart Energy, both read and write access of attributes is required, depending on which clusters are enabled.

Optional Attributes

Many clusters have optional attributes that may be enabled at compile-time via the options header file - for example, the Simple Metering instantaneous demand attribute is enabled in the demonstration application:

```
#define CLD_SM_ATTR_INSTANTANEOUS_DEMAND
```

Note: Cluster-specific compile-time options are detailed in the chapters for the individual clusters in Part II: Smart Energy Clusters. For clusters from the ZCL, refer to the *ZCL User Guide (JN-UG-3077)*.

3.5.2 ZigBee Network Parameters

Smart Energy applications require specific settings of certain ZigBee network parameters. These parameters are set using the ZPS Configuration Editor and the required settings are given in Section 13.5. The full set of ZigBee network parameters are detailed in the *ZigBee PRO Stack User Guide (JN-UG-3048)*.

3.5.3 Building and Loading the Application Binary

A ZigBee PRO Smart Energy application for the JN51xx device is built like any other ZigBee PRO application. The build is normally carried out using the Eclipse IDE. This is the method that we recommend, although it is also possible to use makefiles directly from the command line (Cygwin).

For instructions on building an application in the Eclipse IDE, refer to the *SDK Installation and User Guide (JN-UG-3064)*. This guide also indicates how to load the built application binary file into a JN51xx-based node using the JN51xx Flash Programmer launched from within Eclipse. Alternatively, you can use the JN51xx Flash Programmer directly. In either case, you will need to refer to the *JN51xx Flash Programmer User Guide (JN-UG-3007)* as part of this procedure.

4. Smart Energy Application Coding

This chapter covers general aspects of SE application coding, including essential SE programming concepts, code initialisation, callback functions, reading and writing attributes, and event handling. Application coding that is particular to individual clusters is described later, in the relevant cluster-specific chapter.

Note: SE API functions referenced in this chapter are fully detailed in Chapter 12. Referenced ZCL functions are described in the *ZCL User Guide (JN-UG-3077)*.

4.1 SE Programming Concepts

This section describes the essential programming concepts that are needed in SE application development. The basic operations in an SE network are concerned with reading and setting the attribute values of the clusters of a device.

4.1.1 Shared Device Structures

In each SE device, attribute values are exchanged between the application and the SE library by means of a shared structure. This structure is protected by a mutex (described in the *ZCL User Guide (JN-UG-3077)*). The structure for a particular SE device contains structures for the clusters supported by that device (see Section 2.4). The available device structures are detailed in Section 13.2.

Note: In order to use a cluster which is supported by a device, the relevant option for the cluster must be specified at build-time - see Section 3.5.1.

A shared device structure may be used in either of the following ways:

- The local application writes attribute values to the structure, allowing the ZigBee Cluster Library (ZCL) to respond to commands relating to these attributes. For example, a Metering Device application writes energy consumption data to the local Metering structure and this data is subsequently read remotely by the utility company.
- The ZCL parses incoming commands that write attribute values to the structure. The written values can then be read by the local application. For example, data is remotely written to an IPD structure by the ESP application and the IPD application then reads this data to display it on a screen.

Remote read and write operations involving a shared device structure are illustrated in Figure 10 below. For more detailed descriptions of these operations, refer to Section 4.5 and Section 4.6.

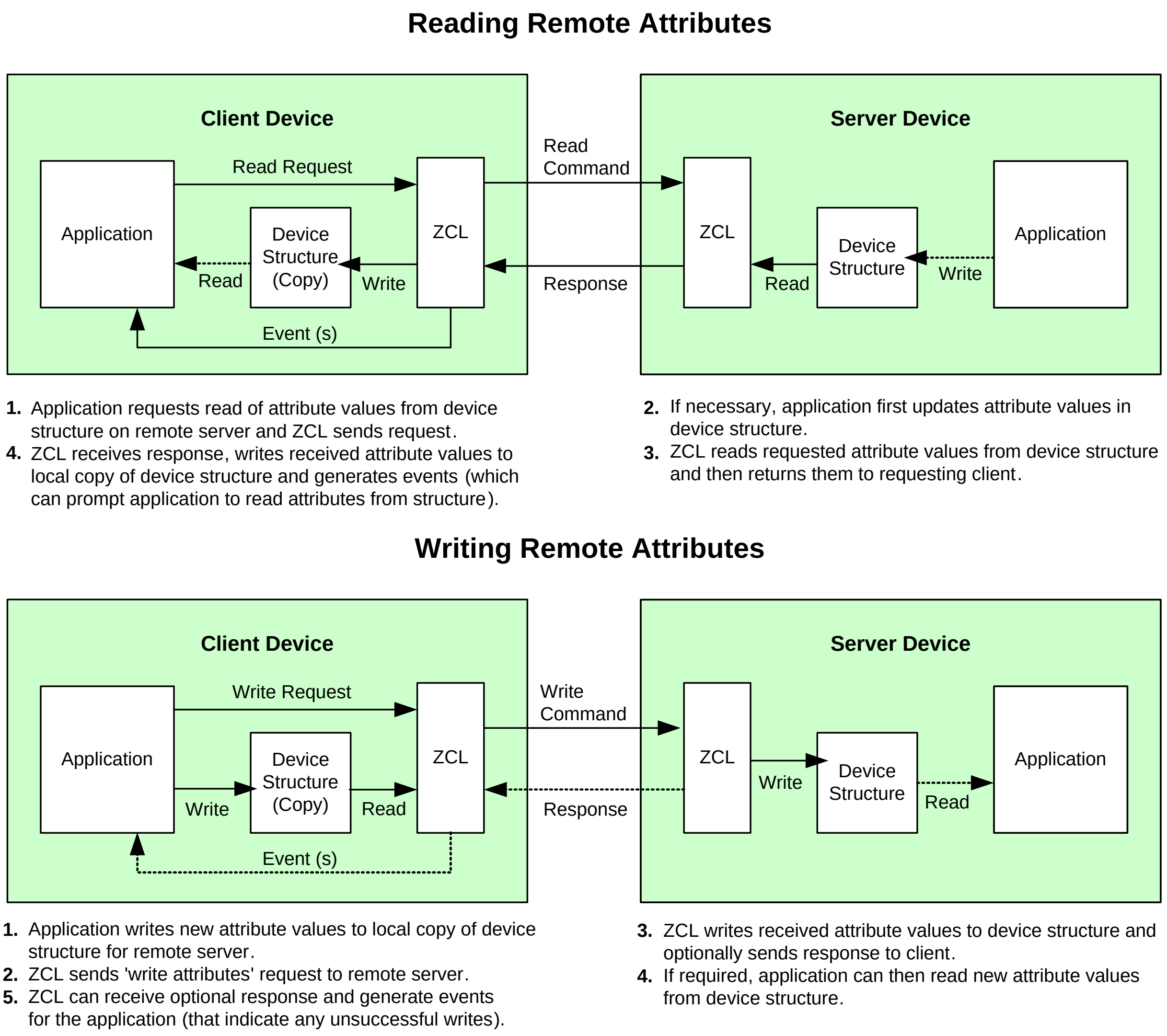
Reading Remote Attributes
Client Device
Application
Read Request
Device Structure (Copy)
ZCL
Read
Write
Event (s)
Read Command
Response
Server Device
ZCL
Device Structure
Application
Read
Write
1. Application requests read of attribute values from device structure on remote server and ZCL sends request.
4. ZCL receives response, writes received attribute values to local copy of device structure and generates events (which can prompt application to read attributes from structure).
2. If necessary, application first updates attribute values in device structure.
3. ZCL reads requested attribute values from device structure and then returns them to requesting client.
Writing Remote Attributes
Client Device
Application
Write Request
Device Structure (Copy)
ZCL
Write
Read
Event (s)
Write Command
Response
Server Device
ZCL
Write
Device Structure
Read
Application
1. Application writes new attribute values to local copy of device structure for remote server.
2. ZCL sends 'write attributes' request to remote server.
5. ZCL can receive optional response and generate events for the application (that indicate any unsuccessful writes).
3. ZCL writes received attribute values to device structure and optionally sends response to client.
4. If required, application can then read new attribute values from device structure.

Figure 10: Operations using Shared Device Structure

Note: Provided that there are no remote attribute writes, the attributes of a cluster server (in the shared structure) on a device are maintained by the local application(s). The equivalent attributes of a cluster client on another device are copies of these cluster server attributes (remotely read from the server).

4.1.2 Addressing

Communications between devices in an SE network are performed using standard ZigBee PRO mechanisms. A brief summary is provided below.

In order to perform an operation (e.g. a read) on a remote node in a ZigBee PRO network, a command must be sent from the relevant output (or client) cluster on the local node to the relevant input (or server) cluster on the remote node.

At a higher level, an application (and therefore the SE device and supported clusters) is associated with a unique endpoint, which acts as the I/O port for the application on the node. Therefore, a command is sent from an endpoint on the local node to the relevant endpoint(s) on the remote node.

The destination node(s) and endpoint(s) must be identified by the sending application. The endpoints on each node are numbered from 1 to 240. The target node(s) can be addressed in a number of different ways, listed below.

- 64-bit IEEE/MAC address
- 16-bit ZigBee network (short) address
- 16-bit group address, relating to a pre-specified group of nodes and endpoints
- A binding, where the source endpoint has been pre-bound to the remote node(s) and endpoint(s)
- A broadcast, in which the message is sent to all nodes of a certain type, one of:
  - only Co-ordinator and Router nodes
  - all End Devices
  - only End Devices for which the radio receiver stays on when they are idle

A destination address structure, `tsZCL_Address`, is defined in the ZCL and is detailed in the *ZCL User Guide (JN-UG-3077)*. Enumerations are provided for the addressing mode and broadcast mode in this structure, and are also detailed in the above manual.

4.1.3 OS Resources

The SE library and ZCL require OS resources, such as tasks and mutexes. These resources are provided by JenOS (Jennic Operating System), supplied in the SDK.

The JenOS resources for an application are allocated using the JenOS Configuration Editor, which is provided as an NXP-specific plug-in for the Eclipse IDE. Use of the JenOS Configuration Editor for an SE application should be based on the ZigBee PRO Smart Energy template or demonstration application (rather than on the standard ZigBee PRO stack template) to ensure that the extra JenOS resources required by the SE profile and the ZCL are available.

A JenOS mutex protects the shared structure that holds the cluster attribute values for a device (see Section 4.1.1 above). This mutex is part of the JenOS Smart Energy template. The ZCL invokes an application callback function to lock and unlock this mutex. The mutex should be used in conjunction with the counting mutex code provided in the appendix of the *ZCL User Guide (JN-UG-3077)*. The software for this mutex operation is contained in the Smart Energy demonstration application.

The task that the SE library and ZCL use to process incoming messages is defined in the JenOS Smart Energy template. Callbacks from the SE library and ZCL to the application will be in the context of this task. The Smart Energy demonstration application and template have a separate task for the user application code. This task also links to the shared-structure mutex in the JenOS configuration so that it can use critical sections to protect access to the shared structures.

Only data events addressed to the correct ZigBee profile, endpoint and cluster are processed by the ZCL, possibly with the aid of a callback function. Stack and data events that are not addressed to an SE endpoint are handled by the application through a callback function. All events are first passed into the ZCL using the function **vZCL_EventHandler()**. The ZCL either processes the event or passes it to the application, invoking the relevant callback function (refer to Section 4.3 for information on callback functions and to Section 4.7 for more details on event handling).

If the ZCL consumes a data event, it will free the corresponding Protocol Data Unit (PDU), otherwise it is the responsibility of the application to free the PDU.

4.2 Initialisation

A ZigBee PRO Smart Energy application is initialised like a normal ZigBee PRO application, as described in the section “Forming a Network” of the *ZigBee PRO Stack User Guide (JN-UG-3048)*. In addition, some SE initialisation must be performed in the application code.

The SE initialisation functions mentioned below must be called after calling **ZPS_eAplAfInit()** and before calling **ZPS_eAplZdoStartStack()**:

1. First initialise the SE library and ZCL using the function **eSE_Initialise()**. This function requires you specify a user-defined callback function for handling stack events (see Section 4.3 below), as well as a pool of APDUs (Application Protocol Data Units) for sending and receiving data.
2. Now set up the SE device(s) handled by your code. Each SE device on the node must be allocated a unique endpoint (in the range 1-240). In addition, its device structure must be registered, as well as a user-defined callback function that will be invoked by the SE library when an event occurs relating to the endpoint (see Section 4.3 below). All of this is done using a registration function for the SE device type - for example, in the case of an IPD, the required function is **eSE_RegisterIPDEndPoint()**.

Note: The set of endpoint registration functions for the different SE device types are detailed in Chapter 12. Functions are provided for a combined ESP/Metering Device and for separate ESP and Metering Device.

4.3 Callback Functions

Two types of user-defined callback function must be provided (and registered as described in Section 4.2):

- **Endpoint Callback Function:** A callback function must be provided for each endpoint used, where this callback function will be invoked when an event occurs (such as an incoming message) relating to the endpoint. The callback function is registered with the SE library when the endpoint is registered using the registration function for the SE device type that the endpoint supports - for example, using **eSE_RegisterIPDEndPoint()** for an IPD (see Chapter 12).
- **General Callback Function:** Events that do not have an associated endpoint are delivered via a callback function that is registered with the SE library through the function **eSE_Initialise()**. For example, stack leave and join events can be received by the application through this callback function.

The endpoint callback function and general callback function both have the type definition given below:

**typedef void (* tfpZCL_ZCLCallBackFunction)
(tsZCL_CallBackEvent ****pCallBackEvent***);**

The callback events are detailed in the *ZCL User Guide (JN-UG-3077)* and event handling is further described in Section 4.7.

4.4 Discovering Endpoints and Clusters

In order to communicate, a cluster client and cluster server must discover and store each other's contact details - that is, the address of the node and the number of the endpoint on which the relevant cluster resides.

The SE application on a node can discover other nodes in the network by calling the ZigBee PRO API function **ZPS_eAplZdpMatchDescRequest()**, which sends out a match descriptor request (as a broadcast to all network nodes or as unicasts to selected nodes). This function allows nodes to be selectively discovered by looking for specific criteria in the Simple Descriptors of the endpoints on the recipient nodes. These criteria include a list of required input (server) clusters and a list of required output (client) clusters. In this way, an application which supports a particular cluster server or client can discover its cluster counterpart(s) in the rest of the network.

If a recipient node satisfies the criteria specified in a match descriptor request, it will respond with a match descriptor response. This response contains the network address of the responding node and a list of the node's endpoints that satisfy the required criteria - for example, the endpoints that support the specified cluster(s).

Once a relevant node and endpoint have been identified:

- The function **ZPS_eAplZdpIeeeAddrRequest()** can be used to obtain the IEEE/MAC address of the node and then both addresses can be added to the local Address Map using the function **ZPS_eAplZdoAddAddrMapEntry()**.
- If data packets between the two endpoints are to be encrypted by means of standard ZigBee PRO security then one of the two nodes must initiate a link key request using the function **ZPS_eAplZdoRequestKeyReq()**.
- The node can bind a local endpoint to the remote endpoint using the function **ZPS_eAplZdpBindUnbindRequest()**.

Note: All of the above functions are described in the *ZigBee PRO Stack User Guide (JN-UG-3048)*.

4.5 Reading Attributes

The most common operation in an SE application is to read attributes from a remote device, e.g. an application on an IPD may need to obtain data from a Metering device. Attributes can be read using a general ZCL function or using an SE function which is specific to the target cluster. The cluster-specific functions for reading attributes are covered in the chapters of this manual that describe the supported clusters. Note that read access to cluster attributes must be explicitly enabled at compile-time as described in Section 3.5.1.

The remainder of this section describes the use of the ZCL function **eZCL_SendReadAttributesRequest()** to send a ‘read attributes’ request, although the sequence is similar when using the cluster-specific ‘read attributes’ functions. The resulting activities on the source and destination nodes are outlined below and illustrated in Figure 11. Note that instances of the shared device structure (which contains the relevant attributes) exist on both the source and destination nodes. The events generated from a ‘read attributes’ request are further described in Section 4.7.

1. On Source Node (Client)

The function **eZCL_SendReadAttributesRequest()** is called to submit a request to read one or more attributes on a cluster on a remote node. The information required by this function includes the following:

- Source endpoint (from which the read request is to be sent)
- Address of destination node for request
- Destination endpoint (on destination node)
- Identifier of the cluster containing the attributes [enumerations provided]
- Number of attributes to be read
- Array of identifiers of attributes to be read [enumerations provided]

2. On Destination Node (Server)

On receiving the ‘read attributes’ request, the SE library software on the destination node performs the following steps:

1. Generates an E_ZCL_CBET_READ_REQUEST event for the destination endpoint callback function which, if required, can update the shared device structure that contains the attributes to be read, before the read takes place.
2. Generates an E_ZCL_CBET_LOCK_MUTEX event for the endpoint callback function, which should lock the mutex that protects the shared device structure - for information on mutexes, refer to the *ZCL User Guide (JN-UG-3077)*
3. Reads the relevant attribute values from the shared device structure and creates a ‘read attributes’ response message containing the read values.
4. Generates an E_ZCL_CBET_UNLOCK_MUTEX event for the endpoint callback function, which should now unlock the mutex that protects the shared device structure (other application tasks can now access the structure).
5. Sends the ‘read attributes’ response to the source node of the request.

3. On Source Node (Client)

On receiving the ‘read attributes’ response, the SE library software on the source node performs the following steps:

1. Generates an E_ZCL_CBET_LOCK_MUTEX event for the source endpoint callback function, which should lock the mutex that protects the relevant shared device structure on the source node.
2. Writes the new attribute values to the shared device structure on the source node.
3. Generates an E_ZCL_CBET_UNLOCK_MUTEX event for the endpoint callback function, which should now unlock the mutex that protects the shared device structure (other application tasks can now access the structure).
4. For each attribute listed in the ‘read attributes’ response, it generates an E_ZCL_CBET_READ_INDIVIDUAL_ATTRIBUTE_RESPONSE message for the source endpoint callback function, which may or may not take action on this message.
5. On completion of the parsing of the ‘read attributes’ response, it generates a single E_ZCL_CBET_READ_ATTRIBUTES_RESPONSE message for the source endpoint callback function, which may or may not take action on this message.

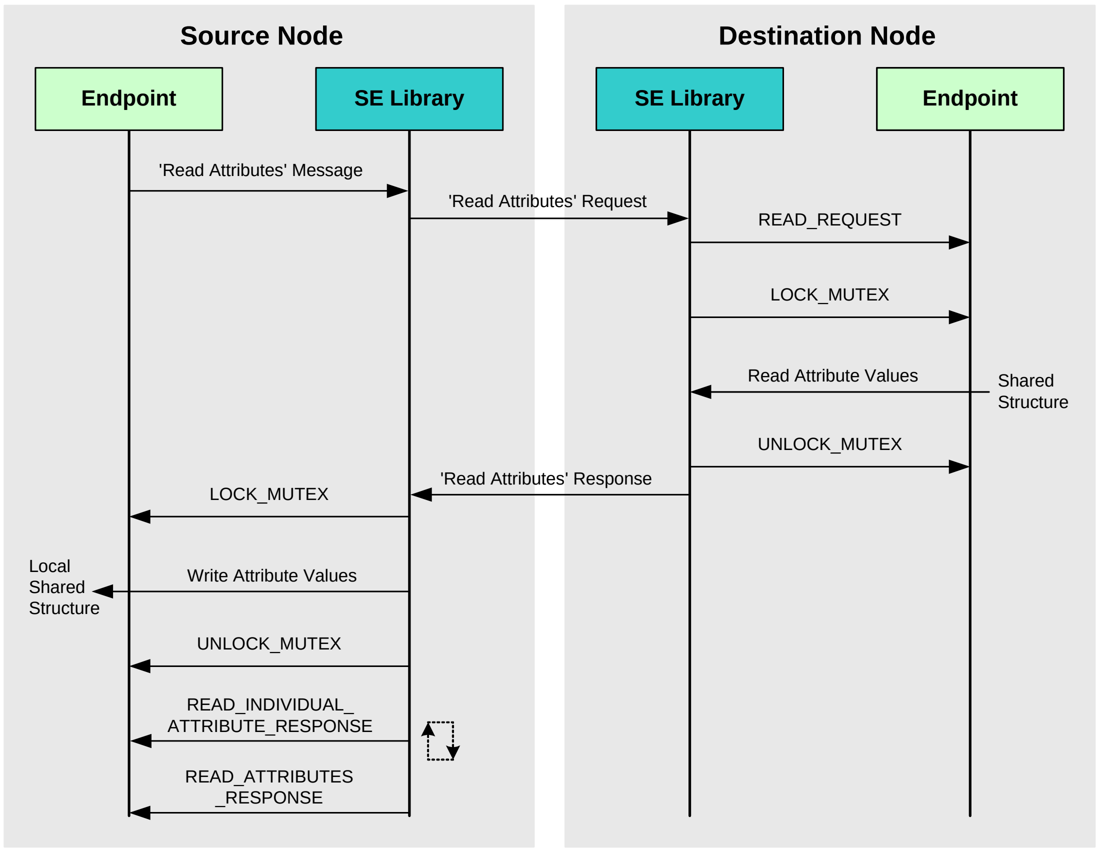

Figure 11: ‘Read Attributes’ Request and Response

Note: The ‘read attributes’ requests and responses arrive at their destinations as data messages. Such a message triggers a stack event of the type ZPS_EVENT_APS_DATA_INDICATION, which is handled as described in Section 4.7.

4.6 Writing Attributes

The ability to write attribute values to a remote cluster is required by some SE devices - for example, an ESP may need to write attributes to a Load Control Device (e.g to configure the device group). Normally, a 'write attributes' request is sent from a client cluster to a server cluster, where the relevant attributes in the shared device structure are updated. Note that write access to cluster attributes must be explicitly enabled at compile-time as described in Section 3.5.1.

Three 'write attributes' functions are provided in the ZCL:

- **eZCL_SendWriteAttributesRequest():** This function sends a 'write attributes' request to a remote device, which attempts to update the attributes in its shared structure. The remote device generates a 'write attributes' response to the source device, indicating success or listing error codes for any attributes that it could not update.
- **eZCL_SendWriteAttributesNoResponseRequest():** This function sends a 'write attributes' request to a remote device, which attempts to update the attributes in its shared structure. However, the remote device does not generate a 'write attributes' response, regardless of whether there are errors.
- **eZCL_SendWriteAttributesUndividedRequest():** This function sends a 'write attributes' request to a remote device, which checks that all the attributes can be written to without error:
  - If all attributes can be written without error, all the attributes are updated.
  - If any attribute is in error, all the attributes are left at their existing values.

  The remote device generates a 'write attributes' response to the source device, indicating success or listing error codes for attributes that are in error.

The activities surrounding a 'write attributes' request on the source and destination nodes are outlined below and illustrated in Figure 12. Note that instances of the shared device structure (which contains the relevant attributes) must be maintained on both the source and destination nodes. The events generated from a 'write attributes' request are further described in Section 4.7.

1. On Source Node (Client)

In order to send a 'write attributes' request, the application on the source node performs the following steps:

1. Locks the mutex that protects the local instance of the shared device structure that contains the attributes to be updated - for information on mutexes, refer to the *ZCL User Guide (JN-UG-3077).*
2. Writes one or more updated attribute values to the local instance of the shared device structure.
3. Unlocks the mutex that protects the local instance of the shared device structure.

4. Calls one of the above ZCL ‘write attributes’ functions to submit a request to update the relevant attributes on a cluster on a remote node. The information required by this function includes the following:
   - Source endpoint (from which the write request is to be sent)
   - Address of destination node for request
   - Destination endpoint (on destination node)
   - Identifier of the cluster containing the attributes [enumerations provided]
   - Number of attributes to be written
   - Array of identifiers of attributes to be written [enumerations provided]

   From the above information, the function is able to pick up the relevant attribute values from the local instance of the shared structure and incorporate them in the message for the remote node.

2. On Destination Node (Server)

On receiving the ‘write attributes’ request, the SE library software on the destination node performs the following steps:

1. For each attribute in the ‘write attributes’ request, generates an E_ZCL_CBET_CHECK_ATTRIBUTE_RANGE event for the destination endpoint callback function. If required, the callback function can do either or both of the following:
   - check that the new attribute value is in the correct range - if the value is out-of-range, the function should set the `eAttributeStatus` field of the event to E_ZCL_ERR_ATTRIBUTE RANGE
   - block the write by setting the the `eAttributeStatus` field of the event to E_ZCL_DENY_ATTRIBUTE_ACCESS

   In the case of an out-of-range value or a blocked write, there is no further processing for that particular attribute following the ‘write attributes’ request.
2. Generates an E_ZCL_CBET_LOCK_MUTEX event for the endpoint callback function, which should lock the mutex that protects the relevant shared device structure - for more on mutexes, refer to the *ZCL User Guide (JN-UG-3077).*
3. Writes the relevant attribute values to the shared device structure - an E_ZCL_CBET_WRITE_INDIVIDUAL_ATTRIBUTE event is generated for each individual attempt to write an attribute value, which the endpoint callback function can use to keep track of the successful and unsuccessful writes.

   Note that if an ‘undivided write attributes’ request was received, an individual failed write will render the whole update process unsuccessful.
4. Generates an E_ZCL_CBET_WRITE_ATTRIBUTES event to indicate that all relevant attributes have been processed and, if required, creates a ‘write attributes’ response message for the source node.
5. Generates an E_ZCL_CBET_UNLOCK_MUTEX event for the endpoint callback function, which should now unlock the mutex that protects the shared device structure (other application tasks can now access the structure).
6. If required, sends a ‘write attributes’ response to the source node of the request.

3. On Source Node (Client)

On receiving an optional 'write attributes' response, the SE library software on the source node performs the following steps:

1. For each attribute listed in the 'write attributes' response, it generates an E_ZCL_CBET_WRITE_INDIVIDUAL_ATTRIBUTE_RESPONSE message for the source endpoint callback function, which may or may not take action on this message. Only attributes for which the write has failed are included in the response and will therefore result in one of these events.
2. On completion of the parsing of the 'write attributes' response, it generates a single E_ZCL_CBET_WRITE_ATTRIBUTES_RESPONSE message for the source endpoint callback function, which may or may not take action on this message.

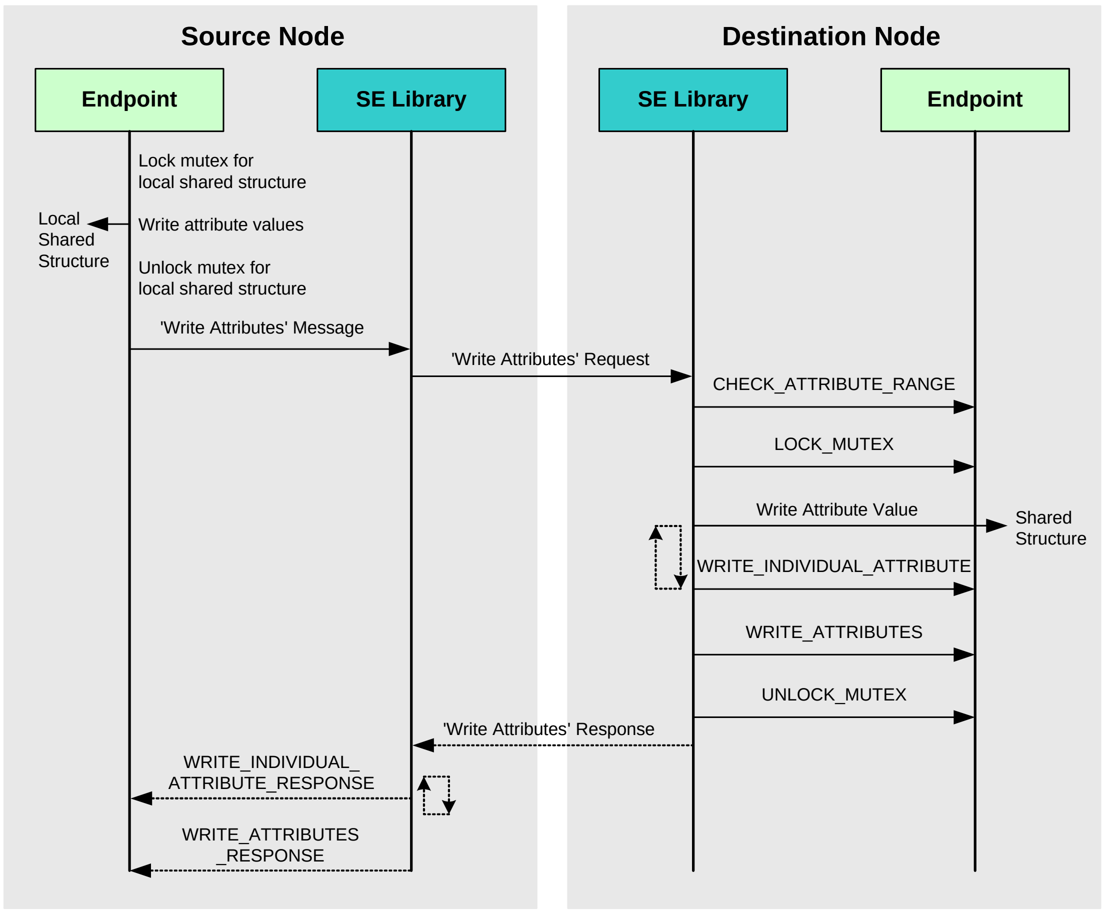
Source Node
Destination Node
Endpoint
SE Library
SE Library
Endpoint
Lock mutex for local shared structure
Local Shared Structure
Write attribute values
Unlock mutex for local shared structure
'Write Attributes' Message
'Write Attributes' Request
CHECK_ATTRIBUTE_RANGE
LOCK_MUTEX
Write Attribute Value
Shared Structure
WRITE_INDIVIDUAL_ATTRIBUTE
WRITE_ATTRIBUTES
UNLOCK_MUTEX
'Write Attributes' Response
WRITE_INDIVIDUAL_ ATTRIBUTE_RESPONSE
WRITE_ATTRIBUTES _RESPONSE

Figure 12: 'Write Attributes' Request and Response

Note: The 'write attributes' requests and responses arrive at their destinations as data messages. Such a message triggers a stack event of the type ZPS_EVENT_APS_DATA_INDICATION, which is handled as described in Section 4.7.

4.7 Handling Events

This section outlines the event handling framework which allows a ZigBee PRO Smart Energy application to deal with stack-related and timer-related events. A stack event is triggered by a message arriving in a message queue and a timer event is triggered when a JenOS timer expires.

The event handling framework for ZigBee PRO Smart Energy is provided by the ZCL. The event must be wrapped in a `tsZCL_CallBackEvent` structure by the application, which then passes this event structure into the ZCL using the function **vZCL_EventHandler()**. The ZCL processes the event and, if necessary, invokes the relevant endpoint callback function. This event structure and event handler function are detailed in the *ZCL User Guide (JN-UG-3077)*, which also provides more details of event processing.

The events that are not cluster-specific are divided into four categories, as shown in Table 9 below - these events are described in the *ZCL User Guide (JN-UG-3077)*. Cluster-specific events are covered in the chapter for the relevant cluster.

Category	Event
Input Events	E_ZCL_ZIGBEE_EVENT
	E_ZCL_CBET_TIMER
Read Events	E_ZCL_CBET_READ_REQUEST
	E_ZCL_CBET_READ_INDIVIDUAL_ATTRIBUTE_RESPONSE
	E_ZCL_CBET_READ_ATTRIBUTES_RESPONSE
Write Events	E_ZCL_CBET_CHECK_ATTRIBUTE_RANGE
	E_ZCL_CBET_WRITE_INDIVIDUAL_ATTRIBUTE
	E_ZCL_CBET_WRITE_ATTRIBUTES
	E_ZCL_CBET_WRITE_INDIVIDUAL_ATTRIBUTE_RESPONSE
	E_ZCL_CBET_WRITE_ATTRIBUTES_RESPONSE
General Events	E_ZCL_CBET_LOCK_MUTEX
	E_ZCL_CBET_UNLOCK_MUTEX
	E_ZCL_CBET_DEFAULT_RESPONSE
	E_ZCL_CBET_UNHANDLED_EVENT
	E_ZCL_CBET_ERROR

Table 9: Events (Not Cluster-Specific)

Note: ZCL error events and default responses may be generated when problems occur in receiving commands. The possible ZCL status codes contained in the events and responses are detailed in the *ZCL User Guide (JN-UG-3077)*.

Part II:
Smart Energy Clusters

5. ZCL Clusters

This chapter provides SE-specific implementation details of the following clusters, which are fully described in the *ZigBee Cluster Library (ZCL) User Guide (JN-UG-3077)*:

- Basic cluster - see Section 5.1
- Time cluster - see Section 5.2
- OTA Upgrade cluster - see Section 5.3

There are no SE-specific recommendations for the other supported ZCL clusters, listed in Appendix A.

5.1 Basic Cluster

The Basic cluster is a mandatory cluster for all ZigBee SE devices and has a Cluster ID of 0x0000. It is a Server-side (input) cluster, so is able to store attributes and respond to commands relating to these attributes. However, NXP have also implemented the Basic client-side (output) cluster on the IPD, to allow this device to read the Basic cluster attributes on the ESP.

5.1.1 Compile-Time Options

The Basic cluster is enabled by defining CLD_BASIC in the **zcl_options.h** file - see Section 3.5.1. In addition, to enable the cluster as a client or server or both, it is also necessary to add one or both of the following to the same file:

```
#define BASIC_CLIENT
#define BASIC_SERVER
```

Other compile-time options are also available for the Basic cluster and are described in the *ZCL User Guide (JN-UG-3077)*.

5.1.2 Mandatory Attributes

The `tsCLD_Basic` structure and attributes of the Basic cluster are fully described in the *ZCL User Guide (JN-UG-3077)*. The cluster contains only two mandatory attributes, the remaining attributes being optional. The two mandatory attributes are outlined below:

- **u8ZCLVersion**: This is an 8-bit version number for the ZCL release that all clusters on the local endpoint(s) conform to.
- **ePowerSource**: This is an 8-bit value in which seven bits indicate the primary power source for the device (e.g. battery) and one bit indicates whether there is a secondary power source for the device.

A set of enumerated values for the full range of possible primary power sources are provided in the structure `teCLD_BAS_PowerSource` of the ZCL software.

The application must set the values of the mandatory `u8ZCLVersion` and `ePowerSource` fields of the Basic cluster structure so that other devices can read them. This should be done immediately after calling the endpoint registration function for the device, e.g. **eSE_RegisterIPDEndPoint()**. Example settings for SE are:

On a mains-powered ESP/Meter:

```
sMeter.sBasicCluster.u8ZCLVersion = 0x01;
sMeter.sBasicCluster.ePowerSource = E_CLD_BAS_PS_SINGLE_PHASE_MAINS;
```

On a battery-powered IPD:

```
sIPD.sLocalBasicCluster.u8ZCLVersion = 0x01;
sIPD.sLocalBasicCluster.ePowerSource = E_CLD_BAS_PS_BATTERY;
```

Note: Since NXP implement the Basic cluster as a client as well as a server on the IPD, there are two Basic cluster structures on this device - one for the local server attributes and another for keeping copies of remote server attribute values. The above settings must be made in the 'local' server structure.

5.2 Time Cluster

The Time cluster is used to maintain a time reference for the transactions in a Smart Energy network and to time-synchronise the SE devices. It has a Cluster ID of 0x000A.

The Time cluster is required in SE devices as indicated in the table below.

	Server-side	Client-side
Mandatory in...	ESP	PCT Load Control Device Smart Appliance
Optional in...		Metering Device IPD

Table 10: Time Cluster in SE Devices

The ESP implements the Time cluster as a server and acts as the time-master for an SE network. The other SE devices in the network implement the Time cluster as a client and time-synchronise with the server.

5.2.1 Compile-Time Options

The Time cluster is enabled by defining CLD_TIME in the **zcl_options.h** file - see Section 3.5.1. In addition, to enable the cluster as a client or server or both, it is also necessary to add one or both of the following to the same file:

```
#define TIME_CLIENT
#define TIME_SERVER
```

Other compile-time options are also available for the Time cluster and are described in the *ZCL User Guide (JN-UG-3077)*.

5.2.2 Time Standards

UTC Time

The Time cluster contains an attribute for the current time, which is referenced to UTC (Co-ordinated Universal Time) and based on the type **UTCTime**, which is defined in the ZigBee standard as:

"UTCTime is an unsigned 32 bit value representing the number of seconds since 0 hours, 0 minutes, 0 seconds, on the 1st of January, 2000 UTC".

ZCL Time

The ZCL also keeps its own time, ‘ZCL time’, which is based on the above **UTCTime** definition. This time is derived from a one-second timer provided by JenOS and is used to drive any ZCL timers that have been registered.

5.2.3 Mandatory Attributes

The `tsCLD_Time` structure and attributes of the Time cluster are fully described in the *ZCL User Guide (JN-UG-3077)*. The cluster contains only two mandatory attributes, the remaining attributes being optional. The two mandatory attributes are outlined below:

- **utctTime**: This is a 32-bit attribute which holds the current time (UTC). On the time-master (ESP), this attribute value is incremented once per second. On all other SE devices, it is the responsibility of the local application to synchronise this time with the time-master. For more information on time-synchronisation, refer to Section 5.2.4.
- **u8TimeStatus**: This is an 8-bit attribute containing the following bitmap:

Bits	Meaning	Description
0	Master	1: Time-master for network 0: Not time-master for network
1	Synchronised (servers only)	1: Server synchronised to another ZigBee SE device 0: Server not synchronised to another ZigBee SE device
2	Master for Time Zone and DST *	1: Master for time-zone and DST 0: Not master for time-zone and DST
3-7	Reserved	-

Table 11: u8TimeStatus Bitmap

* DST= Daylight Saving Time

This attribute must be set as follows on the ESP, on which the Time cluster is a server and which acts as the time-master for the network:

- The ‘Master’ bit should initially be zero until the current time has been obtained from the utility company or from another external time-of-day source. Once the time has been obtained and set, the ‘Master’ bit should be set (to ‘1’).
- The ‘Synchronised’ bit must always be zero, as the ESP does not obtain its time from another SE device within the ZigBee network (this bit is set to ‘1’ only for a secondary Time cluster server that is synchronised to the time-master).
- The ‘Master for Time Zone and DST’ bit must be set (to ‘1’) once the time-zone and Daylight Saving Time (DST) attributes (see below) have been correctly set for the device.

Macros are provided for setting the individual bits of the bitmap - these macros are defined in the header file **time.h** and listed in the *ZCL User Guide (JN-UG-3077)*.

The optional attributes of the Time cluster are mainly concerned with the time-zone and daylight saving.

5.2.4 Time-Synchronisation of Devices

Devices in an SE wireless network need to be time-synchronised (so that they all refer to the same time). Normally, the ESP acts as the time-master from which the other devices set their time, since this device is linked to the utility company from where the master time is obtained.

There are two times on a device that should be maintained during the synchronisation process:

- Time attribute of the Time cluster (`utctTime` field of `tsCLD_Time` structure)
- ZCL time

On the time-master, these times are initialised by the local application using the current time from the utility company and are subsequently maintained using a local one-second timer (see Section 5.2.4.1), as well as occasional re-synchronisations with the utility company.

On all other devices, these times are initialised by the local application by synchronising with the time-master (see Section 5.2.4.2). The ZCL time is subsequently maintained using a local one-second timer and both times are occasionally re-synchronised with the time master (see Section 5.2.4.3).

Synchronisation with the time-master is normally performed via the Time cluster (but can alternatively be performed using a field of the Publish Price command - see Section 6.6).

***Caution:** If there is more than one Time cluster server in the network, devices should only attempt to synchronise to one server in order to prevent their clocks from repeatedly jittering backwards and forwards.*

Note: Some SE clusters use the ZCL time in order to generate events at particular times. When the ZCL is initialised on a device, the ZCL time is not set. Until this time is set, events that depend on the current time (such as a Price event with a ‘start-time of now’) cannot be processed - see Section 5.2.4.1 below.

The diagram in Figure 13 below provides an overview of the time initialisation and synchronisation processes described in the sub-sections that follow.

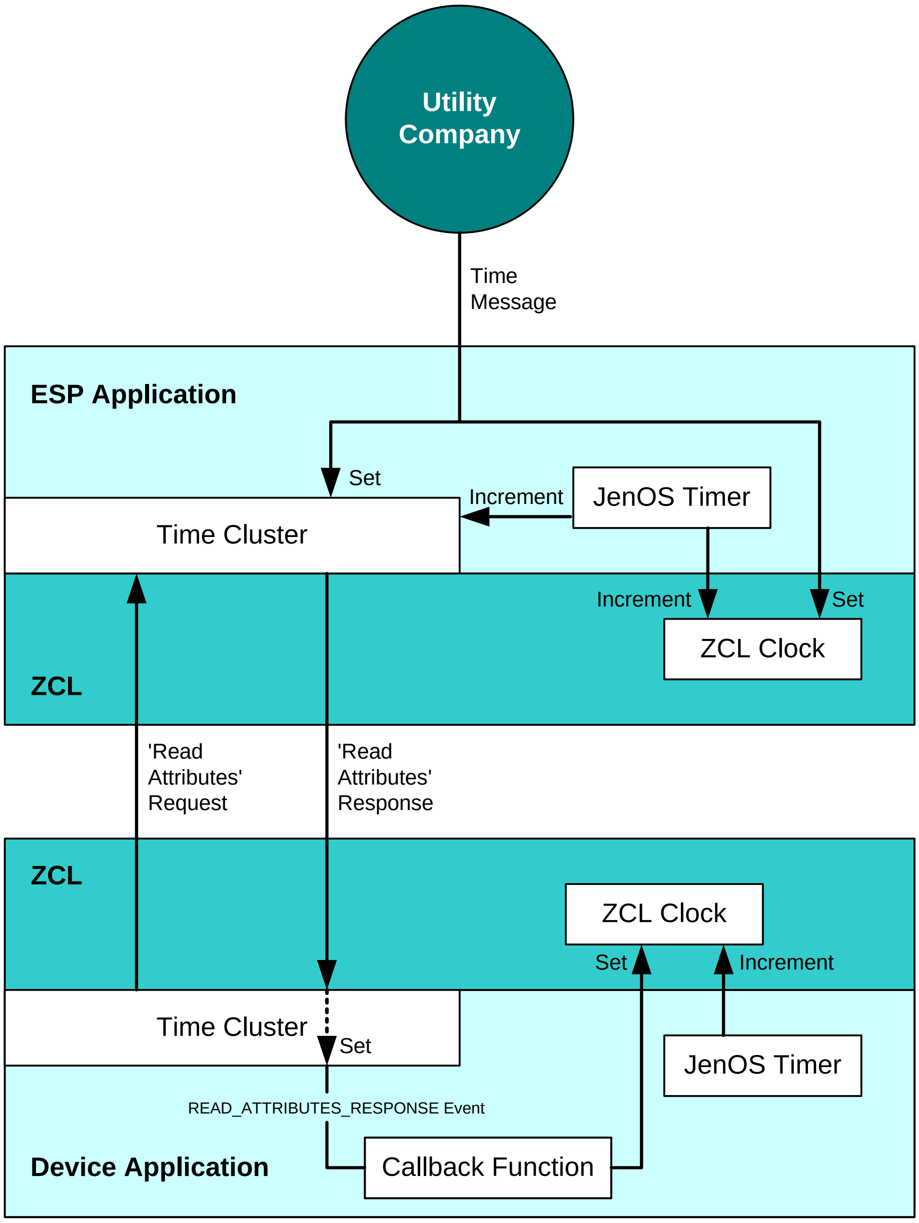

Figure 13: Time Initialisation and Synchronisation

5.2.4.1 Initialising and Maintaining Master Time

The ESP initially obtains the current time (UTC) from the utility company via the backhaul network. The ESP application must use this time to set its ZCL time by calling the function **vZCL_SetUTCTime()** and to set the value of the Time cluster attribute `utctTime` in the local `tsCLD_Time` structure within the shared device structure (securing access with a mutex). The application must also set (to '1') the 'Master' bit of the `u8TimeStatus` attribute of the `tsCLD_Time` structure, to indicate that this device is the time-master and that the time has been set.

Note: The 'Synchronised' bit of the `u8TimeStatus` attribute should always be zero on the time-master, as this device does not synchronise to any other SE device within the ZigBee network.

If the ESP has also obtained time-zone and daylight saving information from the utility company (or has been pre-programmed with this information), the ESP application must set (to '1') the 'Master for Time Zone and DST' bit of the `u8TimeStatus` attribute and write the relevant optional attributes. These optional attributes on the ESP can then be used to provide time-zone and daylight saving information to other devices.

Note: The ESP can prevent other devices from attempting to read its Time cluster attributes before the time has been set - the initial synchronisation with the utility company should be done after calling the relevant endpoint registration function (for example, **eSE_RegisterEspEndPoint()**) but before calling **ZPS_eAplZdoStartStack()**.

The ZCL time and the `utctTime` attribute are subsequently incremented from a local one-second timer provided by JenOS, as follows. On expiration of the JenOS timer, an event is generated (from the hardware/software timer that drives the JenOS timer), which causes JenOS to activate a ZCL user task. The event is initially handled by this task as described in Section 4.7, resulting in an E_ZCL_CBET_TIMER event being passed to the ZCL via the function **vZCL_EventHandler()**. The following actions should then be performed:

1. The ZCL automatically increments the ZCL time and may run cluster-specific schedulers (e.g. for maintaining a price list).
2. The user task updates the value of the `utctTime` attribute of the `tsCLD_Time` structure within the shared device structure (securing access with a mutex).
3. The user task resumes the one-second timer using the JenOS function **OS_eContinueSWTimer()**.

The demonstration application in the Application Note *Smart Energy HAN Solutions (JN-AN-1135)* illustrates how to do this.

Both the ZCL time and the `utctTime` attribute must also be updated by the application when a time update is received from the utility company.

5.2.4.2 Initial Synchronisation of Devices

It is the responsibility of the application on an SE device to perform time-synchronisation with the ESP. The application must remotely read the Time cluster attributes from the ESP by calling the function **eSE_ReadTimeAttributes()** or **eZCL_SendReadAttributesRequest()**, which will result in a 'read attributes' response containing the Time cluster data. On receiving this response, a 'data indication' stack event is generated on the local device, which causes JenOS to activate a ZCL user task. The event is initially handled by this task as described in Section 4.7, resulting in an E_ZCL_ZIGBEE_EVENT event being passed to the ZCL via the function **vZCL_EventHandler()**. Provided that the event contains a message incorporating a 'read attributes' response, the ZCL:

1. automatically sets the `utctTime` field of the `tsCLD_Time` structure to the value of the same attribute in the 'read attributes' response (and also sets other Time cluster attributes, if requested)
2. invokes the relevant user-defined callback function (see Section 4.7), which must read the local `utctTime` attribute (securing access with a mutex) and use this value to set the ZCL time by calling the function **vZCL_SetUTCTime()**

The demonstration application in the Application Note *Smart Energy HAN Solutions (JN-AN-1135)* illustrates how to do this.

> **Note:** When a device attempts to time-synchronise with the ESP, it should check the `u8TimeStatus` attribute in the 'read attributes' response. If the 'Master' bit of this attribute is not equal to '1', the obtained time should not be trusted and the time should not be set. The device should wait and try to synchronise again later.

It may also be possible to obtain time-zone and daylight saving information from the ESP. If available, this information will be returned in the 'read attributes' response. However, before using these optional Time cluster attributes from the response, the application should first check that the 'Master for Time Zone and DST' bit of the `u8TimeStatus` attribute is set (to '1') in the response.

The ZCL time and `utctTime` attribute value on the local device are subsequently maintained as described in Section 5.2.4.3.

5.2.4.3 Re-synchronisation of Devices

After the initialisation described in Section 5.2.4.2, the ZCL time must be updated by the application on each one-second tick of the local JenOS timer. The ZCL time is updated from the timer in the same way as described for the time-master (ESP) in Section 5.2.4.1 (except the Time cluster `utctTime` attribute is not updated).

Due to the inaccuracy of the local one-second timer, the ZCL time is likely to lose synchronisation with the master time on the ESP. It will therefore be necessary to occasionally re-synchronise the local ZCL time with the ESP - the `utctTime` attribute value is also updated at the same time. A device re-synchronises with the ESP by first remotely reading the `utctTime` attribute on the ESP using the function **eSE_ReadTimeAttributes()** or **eZCL_SendReadAttributesRequest()**. On receiving the 'read attributes' response from the ESP, the operations performed are the same as those described for initial synchronisation in Section 5.2.4.2.

Note: If a device also implements the Price cluster, time re-synchronisation can be performed using the current time embedded in the Publish Price commands - see Section 6.6. However, these commands do not carry time-zone or daylight saving information. If such a command has not been received for an extended period of time, the device may need to initiate a time re-synchronisation with the ESP as described above.

In order to avoid excessive re-synchronisation traffic across the network, the ZigBee Smart Energy specification states that "*time accuracy on client devices shall be within ±1 minute of the server device (ESP) per 24 hour period*". In addition, the specification demands that clock accuracy on the client devices "*never requires more than one time synchronization event per 24 hour period*". As a general rule, an application should initiate a time re-synchronisation if it has not received any communications that contain a time-stamp in the last 48 hours. However, in the case of a failed synchronisation (see Note in Section 5.2.4.2), a new attempt to synchronise can be made after a much shorter time, as this situation is only likely to occur when the ESP and the device have been powered around the same time.

Note: If the ESP receives a time update from the utility company then the ESP application must update its ZCL time and its time attribute.

5.2.4.4 Re-synchronisation Following Sleep

The above re-synchronisation (in Section 5.2.4.3) is easy to achieve for a device that does not sleep. In the case of a device that sleeps, on waking from sleep, the application should update the ZCL time using the function **vZCL_SetUTCTime()** according to the duration for which the device was asleep. This requires the sleep duration to be timed.

While sleeping, the JN51xx microcontroller normally uses its RC oscillator for timing purposes, which does not maintain the required accuracy for Smart Energy. It is therefore recommended that a more accurate external crystal is used to time the sleep periods.

The **vZCL_SetUTCTime()** function does not cause timer events to be executed. If the device is awake for less than one second, the application should generate a E_ZCL_CBET_TIMER event to prompt the ZCL to run any timer-related functions, such as maintenance of the list of scheduled prices. Note that when passed into **vZCL_EventHandler()**, this event will increment the ZCL time by one second.

5.2.4.5 Checking ZCL Time Synchronisation

In addition to the time-related functions mentioned earlier, the SE API provides the following functions for checking ZCL time synchronisation:

- **u32ZCL_GetUTCTime()** obtains the ZCL time (held locally).
- **bZCL_GetTimeHasBeenSynchronised()** determines whether the device has been time-synchronised.
- **vZCL_ClearTimeHasBeenSynchronised()** can be used to specify that the device can no longer be considered to be synchronised (for example, if there has been a problem in accessing the Time cluster server over a long period).

The above functions provide the means for an application on a device that hosts other time-related clusters (e.g. Price and Messaging) to discover whether the device is time-synchronised with the rest of the network or, in the case of the ESP, with the utility company. If the device is not synchronised, these clusters will be unable to send and receive messages (for more information on the resulting Price cluster issues, refer to Section 6.6).

5.3 OTA Upgrade Cluster

The Over-the-Air (OTA) Upgrade cluster provides the facility to upgrade application software on the nodes of a ZigBee PRO network by distributing the replacement software through the network (over the air) from a designated node. This cluster is not officially a part of the ZCL but is described in the *ZCL User Guide (JN-UG3077)*, since it can be included in any ZigBee application profile. The OTA Upgrade cluster has a Cluster ID of 0x0019.

The OTA Upgrade cluster is optional but is particularly useful in Smart Energy networks, allowing upgrade software obtained from the utility company to be easily distributed to the SE network devices. In this case:

- The ESP normally acts as the OTA Upgrade cluster server, which obtains the upgrade software from the utility company (via the backhaul network) and distributes this software within the SE network.
- Other SE devices act as OTA Upgrade cluster clients, able to receive software images from the server and use this software to update the running application.

5.3.1 Compile-Time Options

The OTA Upgrade cluster is enabled by defining CLD_OTA in the **zcl_options.h** file - see Section 3.5.1. In addition, to enable the cluster as a client or server or both, it is also necessary to add one or both of the following to the same file:

```
#define OTA_CLIENT
#define OTA_SERVER
```

Other compile-time options are also available for the OTA Upgrade cluster and are described in the *ZCL User Guide (JN-UG-3077)*.

5.3.2 Mandatory Attributes

The OTA Upgrade cluster attributes are located only on a cluster client and are contained on the structure `tsZCL_AttributeDefinition` - this structure and the attributes are fully described in the *ZCL User Guide (JN-UG-3077)*. The cluster contains only two mandatory attributes, the remaining attributes being optional. The two mandatory attributes are outlined below:

- **u64UgradeServerID**: Contains the 64-bit IEEE/MAC address of the OTA Upgrade server for the (local) client. In an SE network, this is normally the address of the ESP. This address can be fixed during manufacture or discovered during network formation/operation.
- **u8ImageUpgradeStatus**: Contains the status of the client device in relation to image downloads and upgrades. Refer to the *ZCL User Guide (JN-UG-3077)* for the possible values.

6. Price Cluster

This chapter outlines the Price cluster which is defined in the ZigBee Smart Energy profile, and is used to hold and exchange price information.

The Price cluster has a Cluster ID of 0x0700.

***Important:** While the Price cluster software supports Block mode, this mode is not certifiable in SE 1.1.1 (07-5356-17) or earlier and is therefore not fully documented in this chapter. Customers who wish to use Block mode should contact NXP for direct support.*

6.1 Overview

The Price cluster is required in SE devices as indicated in the table below.

	Server-side	**Client-side**
Mandatory in...	ESP	Smart Appliance
Optional in...		ESP Metering Device IPD PCT Load Control Device

Table 12: Price Cluster in SE Devices

The ESP normally acts as the Price cluster server, holding price information received from the utility company. Other devices act as clients and receive price information from the ESP. The clients' price information must be kept up-to-date with the server's price information.

The Price cluster is enabled by defining CLD_PRICE in the **zcl_options.h** file - see Section 3.5.1. Further compile-time options for the Price cluster are detailed in Section 6.13.

The Price cluster can operate in a mode in which pricing is based on the time at which the consumption occurs - this is called Time-Of-Use (TOU) mode. The cluster allows up to fifteen price 'tiers', numbered 1 to 15, which correspond to different time periods. Each price tier is given a label, which is used to identify the tier - typical labels are "Normal", "Shoulder", "Peak", "Real-time Pricing" and "Critical Peak". The tiers must be numbered consecutively in price order, with Tier 1 being the cheapest. *Note that tiers 7 to 15 are not certifiable in SE 1.1.1 or earlier and are reserved for future use.*

The information that can potentially be stored in the Price cluster is organised into the following attribute sets: Tier Label, Block Threshold, Block Period, Commodity, Block Price Information, Billing Period Information. The attribute sets Block Threshold, Block Period, Block Price Information and Billing Period Information are reserved for future use (with Block mode). There is also a set of attributes exclusively for use on a Price cluster client.

The cluster includes commands for requesting and publishing (distributing) price information. The price information that is valid for a certain time is sent from the Price cluster server (ESP) to the Price cluster clients using **Publish Price** commands, which may be sent from the ESP under the following circumstances:

- Unsolicited from the server - for example, when new pricing information has been received from the utility company or a new price tier becomes active
- In response to a **Get Current Price** command, sent by a client that needs the price for the current time period
- In response to a **Get Scheduled Prices** command, sent by a client that needs both current and future prices

The SE API provides functions for implementing the cluster commands. These functions are referenced in Section 6.4 and Section 6.5, and detailed in Section 6.9.

6.2 Price Cluster Structure and Attributes

The Price cluster is contained in the following `tsCLD_Price` structure:

```
typedef struct CLD_Price_tag
{
    /* Tier Price Label Set (D.4.2.2.1) */
#if (CLD_P_ATTR_TIER_PRICE_LABEL_MAX_COUNT != 0)
    tsZCL_OctetString  asTierPriceLabel[CLD_P_ATTR_TIER_PRICE_LABEL_MAX_COUNT];
    uint8  au8TierPriceLabel[CLD_P_ATTR_TIER_PRICE_LABEL_MAX_COUNT][SE_PRICE_SERVER_MAX_STRING_LENGTH];
#endif

    /* Block Threshold Set (D.4.2.2) */
#if (CLD_P_ATTR_BLOCK_THRESHOLD_MAX_COUNT != 0)
    zuint48            au48BlockThreshold[CLD_P_ATTR_BLOCK_THRESHOLD_MAX_COUNT];
#endif
    /* Block Period Set (D.4.2.2.3) */
#ifdef CLD_P_ATTR_START_OF_BLOCK_PERIOD
    zutctime           utctStartOfBlockPeriod;
#endif

#ifdef CLD_P_ATTR_BLOCK_PERIOD_DURATION
    zuint24            u24BlockPeriodDuration;
#endif

#ifdef CLD_P_ATTR_THRESHOLD_MULTIPLIER
    zuint24            u24ThresholdMultiplier;
#endif

#ifdef CLD_P_ATTR_THRESHOLD_DIVISOR
    zuint24            u24ThresholdDivisor;
#endif

    /* Commodity Set Set (D.4.2.2.4) */
#ifdef CLD_P_ATTR_COMMODITY_TYPE
    zenum8             e8CommodityType;
#endif

#ifdef CLD_P_ATTR_STANDING_CHARGE
    zuint32            u32StandingCharge;
#endif
```

```
#ifdef CLD_P_ATTR_CONVERSION_FACTOR
    zuint32             u32ConversionFactor;
#endif

#ifdef CLD_P_ATTR_CONVERSION_FACTOR_TRAILING_DIGIT
    zbmap8              b8ConversionFactorTrailingDigit;
#endif

#ifdef CLD_P_ATTR_CALORIFIC_VALUE
    zuint32             u32CalorificValue;
#endif

#ifdef CLD_P_ATTR_CALORIFIC_VALUE_UNIT
    zenum8              e8CalorificValueUnit;
#endif

#ifdef CLD_P_ATTR_CALORIFIC_VALUE_TRAILING_DIGIT
    zbmap8              b8CalorificValueTrailingDigit;
#endif

    /* Block Price Information Set (D.4.2.2.5) */
#if (CLD_P_ATTR_NO_TIER_BLOCK_PRICES_MAX_COUNT != 0)
    zuint32             au32NoTierBlockPrice[CLD_P_ATTR_NO_TIER_BLOCK_PRICES_MAX_COUNT];
#endif

#if ((CLD_P_ATTR_NUM_OF_TIERS_PRICE > 0)&&(CLD_P_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_PRICE != 0))
    zuint32             au32Tier1BlockPrice[CLD_P_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_PRICE];
#endif

#if ((CLD_P_ATTR_NUM_OF_TIERS_PRICE > 1)&&(CLD_P_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_PRICE != 0))
    zuint32             au32Tier2BlockPrice[CLD_P_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_PRICE];
#endif

#if ((CLD_P_ATTR_NUM_OF_TIERS_PRICE > 2)&&(CLD_P_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_PRICE != 0))
    zuint32             au32Tier3BlockPrice[CLD_P_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_PRICE];
#endif

#if ((CLD_P_ATTR_NUM_OF_TIERS_PRICE > 3)&&(CLD_P_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_PRICE != 0))
    zuint32             au32Tier4BlockPrice[CLD_P_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_PRICE];
#endif

#if ((CLD_P_ATTR_NUM_OF_TIERS_PRICE > 4)&&(CLD_P_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_PRICE != 0))
    zuint32             au32Tier5BlockPrice[CLD_P_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_PRICE];
#endif

#if ((CLD_P_ATTR_NUM_OF_TIERS_PRICE > 5)&&(CLD_P_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_PRICE != 0))
    zuint32             au32Tier6BlockPrice[CLD_P_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_PRICE];
#endif

#if ((CLD_P_ATTR_NUM_OF_TIERS_PRICE > 6)&&(CLD_P_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_PRICE != 0))
    zuint32             au32Tier7BlockPrice[CLD_P_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_PRICE];
#endif

#if ((CLD_P_ATTR_NUM_OF_TIERS_PRICE > 7)&&(CLD_P_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_PRICE != 0))
    zuint32             au32Tier8BlockPrice[CLD_P_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_PRICE];
#endif

#if ((CLD_P_ATTR_NUM_OF_TIERS_PRICE > 8)&&(CLD_P_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_PRICE != 0))
    zuint32             au32Tier9BlockPrice[CLD_P_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_PRICE];
#endif

#if ((CLD_P_ATTR_NUM_OF_TIERS_PRICE > 9)&&(CLD_P_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_PRICE != 0))
    zuint32             au32Tier10BlockPrice[CLD_P_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_PRICE];
#endif
```

```
#if ((CLD_P_ATTR_NUM_OF_TIERS_PRICE > 10)&&(CLD_P_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_PRICE != 0))
    zuint32             au32Tier11BlockPrice[CLD_P_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_PRICE];
#endif

#if ((CLD_P_ATTR_NUM_OF_TIERS_PRICE > 11)&&(CLD_P_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_PRICE != 0))
    zuint32             au32Tier12BlockPrice[CLD_P_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_PRICE];
#endif

#if ((CLD_P_ATTR_NUM_OF_TIERS_PRICE > 12)&&(CLD_P_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_PRICE != 0))
    zuint32             au32Tier13BlockPrice[CLD_P_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_PRICE];
#endif

#if ((CLD_P_ATTR_NUM_OF_TIERS_PRICE > 13)&&(CLD_P_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_PRICE != 0))
    zuint32             au32Tier14BlockPrice[CLD_P_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_PRICE];
#endif

#if ((CLD_P_ATTR_NUM_OF_TIERS_PRICE > 14)&&(CLD_P_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_PRICE != 0))
    zuint32             au32Tier15BlockPrice[CLD_P_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_PRICE];
#endif

#ifdef CLD_P_ATTR_START_OF_BILLING_PERIOD
    zutctime            utctStartOfBillingPeriod;
#endif

#ifdef CLD_P_ATTR_BILLING_PERIOD_DURATION
    zuint24             u24BillingPeriodDuration;
#endif

#ifdef CLD_P_CLIENT_ATTR_PRICE_INCREASE_RANDOMIZE_MINUTES
    uint8               u8ClientIncreaseRandomize;
#endif

#ifdef CLD_P_CLIENT_ATTR_PRICE_DECREASE_RANDOMIZE_MINUTES
    uint8               u8ClientDecreaseRandomize;
#endif

#ifdef CLD_P_CLIENT_ATTR_COMMODITY_TYPE
    zenum8              e8ClientCommodityType;
#endif

} tsCLD_Price;
```

where:

‘Tier Label’ Attribute Set

- The following are optional attributes that are only relevant to TOU mode (*tiers 7 to 15 are not certifiable in SE 1.1.1 or earlier and are reserved for future use*):
    - `asTierPriceLabel[CLD_P_ATTR_TIER_PRICE_LABEL_MAX_COUNT]` is a `tsZCL_OctetString` structure containing information on tier labels. The maximum size of `asTierPriceLabel` is defined by assigning a value to `CLD_P_ATTR_TIER_PRICE_LABEL_MAX_COUNT`. This optional element is paired with `au8TierPriceLabel` (below)
    - `au8TierPriceLabel[CLD_P_ATTR_TIER_PRICE_LABEL_MAX_COUNT][SE_PRICE_SERVER_MAX_STRING_LENGTH]` is an array containing the tier labels, e.g. "Peak". This optional element is paired with the element `asTierPriceLabel` (above)

Note: Memory space for each (enabled) price tier label is statically allocated and comprises 13 bytes per label (plus one byte for the ‘octet count’). Therefore, memory space remains allocated for unused bytes.

‘Block Threshold’ Attribute Set

- The following are optional attributes that relate to Block mode and are fully described in the *ZigBee Smart Energy Profile Specification* (*these attributes are not certifiable in SE 1.1.1 or earlier and are for future use*):
  - `au48BlockThreshold[CLD_P_ATTR_BLOCK_THRESHOLD_MAX_COUNT]`

‘Block Period’ Attribute Set

- The following are optional attributes that relate to Block mode and are fully described in the *ZigBee Smart Energy Profile Specification* (*these attributes are not certifiable in SE 1.1.1 or earlier and are for future use*):
  - `utctStartOfBlockPeriod`
  - `u24BlockPeriodDuration`
  - `u24ThresholdMultiplier`
  - `u24ThresholdDivisor`

‘Commodity’ Attribute Set

- The following are optional attributes:
  - `e8CommodityType` is an enumeration representing the type of commodity (e.g. gas) to which the prices apply - the enumerations used are those provided in the `teCLD_SM_MeteringDeviceType` structure of the Simple Metering cluster and listed in Section 8.10.6
  - `u32StandingCharge` is the value of a fixed daily 'standing charge' associated with supplying the commodity, expressed in the currency and with the decimal places indicated in the Publish Price command described in Section 6.11.1 (the value 0xFFFFFFFF indicates that the field is not used)
  - `u32ConversionFactor` is used only for gas and accounts for the variation of gas volume with temperature and pressure (and is dimensionless). The Price server can change this conversion factor at any time and this attribute contains the currently active value. The default value is 1. The position of the decimal point is indicated by `b8ConversionFactorTrailingDigit` described below.
  - `b8ConversionFactorTrailingDigit` is an 8-bit bitmap which indicates the location of the decimal point in the `u32ConversionFactor` attribute. The most significant 4 bits indicate the number of digits after the decimal point. The remaining bits are reserved.

- `u32CalorificValue` is used only for gas and indicates the quantity of energy in MJ that is generated per unit volume or unit mass of gas burned (see `e8CalorificValueUnit`) - the value can be used to calculate energy consumption in kWh. The position of the decimal point is indicated by `b8CalorificValueTrailingDigit` described below.
- `e8CalorificValueUnit` is an enumerated value indicating whether `u32CalorificValue` is quantified per unit volume or per unit mass. The possible values are 0x01 for MJ/m^3 and 0x02 for MJ/kg (all other values are reserved).
- `b8CalorificValueTrailingDigit` is an 8-bit bitmap which indicates the location of the decimal point in the `u32CalorificValue` attribute. The most significant 4 bits indicate the number of digits after the decimal point. The remaining bits are reserved.

‘Block Price Information’ Attribute Set

- The following are optional attributes that relate to Block mode and are fully described in the *ZigBee Smart Energy Profile Specification* (*these attributes are not certifiable in SE 1.1.1 or earlier and are for future use*):
  - `au32NoTierBlockPrice[CLD_P_ATTR_NO_TIER_BLOCK_PRICES_MAX_COUNT]`
  - `au32Tier1BlockPrice[CLD_P_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_PRICE]`
  - `au32Tier2BlockPrice[CLD_P_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_PRICE]`
  - `au32Tier3BlockPrice[CLD_P_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_PRICE]`
  - `au32Tier4BlockPrice[CLD_P_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_PRICE]`
  - `au32Tier5BlockPrice[CLD_P_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_PRICE]`
  - `au32Tier6BlockPrice[CLD_P_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_PRICE]`
  - `au32Tier7BlockPrice[CLD_P_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_PRICE]`
  - `au32Tier8BlockPrice[CLD_P_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_PRICE]`
  - `au32Tier9BlockPrice[CLD_P_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_PRICE]`
  - `au32Tier10BlockPrice[CLD_P_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_PRICE]`
  - `au32Tier11BlockPrice[CLD_P_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_PRICE]`
  - `au32Tier12BlockPrice[CLD_P_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_PRICE]`
  - `au32Tier13BlockPrice[CLD_P_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_PRICE]`
  - `au32Tier14BlockPrice[CLD_P_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_PRICE]`
  - `au32Tier15BlockPrice[CLD_P_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_PRICE]`

‘Billing Period Information’ Attribute Set

- The following are optional attributes that relate to Block mode (*both attributes are not certifiable in SE 1.1.1 or earlier and are for future use*):
  - `utctStartOfBillingPeriod`
  - `u24BillingPeriodDuration`

Client Attribute Set

- The following set of attributes are only for use on a Price cluster client:
  - `u8ClientIncreaseRandomize` represents the maximum length of time, in minutes, between a client node applying a price increase and taking a resulting action (such as reducing its power consumption). The action may be performed before or after the price increase is implemented, and the delay (either way) must be chosen at random by the application on the node. The maximum is set in minutes, in the range 0 to 60 minutes, but it is recommended that the random delay is selected in seconds.
  - `u8ClientDecreaseRandomize` represents the maximum length of time, in minutes, between a client node applying a price decrease and taking a resulting action (such as switching itself on). The action may be performed before or after the price decrease is implemented, and the delay (either way) must be chosen at random by the application on the node. The maximum is set in minutes, in the range 0 to 60 minutes, but it is recommended that the random delay is selected in seconds.
  - `e8ClientCommodityType` is an enumeration representing the commodity that is priced on the client device. This enumeration is one from the 'Metering Device Type' enumerations listed in Table 25 on page 219.

Note: Price information for Time-Of-Use (TOU) mode is held in the `tsSE_PricePublishPriceCmdPayload` structure described in Section 6.11.1. Prices are matched to tiers using the strings defined in the Tier Label attributes.

6.3 Attribute Settings

The Price cluster structure (see Section 6.2) contains no mandatory elements. All elements are optional, each being enabled/disabled through a corresponding macro defined in the **zcl_options.h** file - for example, the commodity type attribute is enabled/disabled through the macro CLD_P_ATTR_COMMODITY_TYPE. The attributes that are used will depend on the number of tiers implemented (and Block mode attributes must be disabled).

Note 1: The Tier Label attributes are connected to the tier-related attributes in the Simple Metering cluster, e.g. `u48CurrentTier6SummationDelivered` for Tier 6. For a complete list of these Simple Metering attributes, refer to Section 8.2. SE-compliance testing includes these attributes.

Note 2: The price information for Time-Of-Use (TOU) mode is stored in the structure `tsSE_PricePublishPriceCmdPayload` described in Section 6.11.1.

6.4 Initialising and Maintaining Price Lists

A list of prices is held on both the Price cluster server (ESP) and client(s). The price list on a client must be maintained to mirror the price list on the server. On device start-up, the Price cluster software initialises the device's price list as empty. The price lists are then built and maintained as described below.

The ESP receives price information from the utility company and populates its price list with this information. The application on the ESP does this by calling the function **eSE_PriceAddPriceEntry()** for each new price received from the utility company. This function also sends out a Publish Price command containing the new price information to all Price cluster clients in the network. On receiving this command, a Price cluster client will automatically add this price information to its own price list (see Section 6.5.1). However, at ESP start-up, there may be no other active nodes in the network to receive the Publish Price commands (since the ESP is normally also the ZigBee Co-ordinator and will therefore be the first node to be started). For this reason, the Price cluster clients should normally request the scheduled prices from the ESP when they start up, as described in Section 6.5.3.

Note: When initialising the price list at ESP start-up, the ESP application should call **eSE_PriceAddPriceEntry()** with the address mode parameter set to E_ZCL_AM_NO_TRANSMIT, so that the price additions are not subsequently transmitted.

Note: A Price cluster server should take precautions to prevent clients from attempting to read the server price list during ESP initialisation, before the prices have been received from the utility company. This can be achieved by adding the obtained prices to the server price list after the call to the relevant endpoint registration function (for example, **eSE_RegisterEspEndPoint()**) but before the call **to ZPS_eAplZdoStartStack()**.

A price list is maintained in time order and if there is an active price, this will be positioned at the head of the list (with index 0). Price lists on clients are updated to reflect the price list on the server, as described in Section 6.5.

Note: The Price cluster of ZigBee Smart Energy automatically deletes a price entry from a client or server price list immediately after the price event has expired. This is because the start-time of a price event is a universal time (UTC) and therefore corresponds to a one-off event. In practice, the price list may need a new price schedule daily, which may be provided by the utility company. Alternatively, if a similar schedule is required every day, the ESP application can keep a local copy of the schedule, which it can modify (e.g. start-times) and add to the price list on a daily basis.

The active price is always at the head of the price list (entry zero). The application should check that the entry at the head of the list is active before displaying it as the current price. If it is not active, a message may be displayed indicating that the current price is not known. The item at the head of the list is active if both of the following hold:

- Its start time is less than or equal to the current time, obtained by **u32ZCL_GetUTCTime()**
- The time on the client has been synchronised, i.e. a call to **bZCL_GetTimeHasBeenSynchronised()** returns TRUE

In addition to the function **eSE_PriceAddPriceEntry()**, the SE API contains other functions that allow an ESP application to access and manipulate its price list:

- **eSE_PriceGetPriceEntry()** obtains the price entry with the specified index
- **eSE_PriceDoesPriceEntryExist()** checks whether there is a price entry with the specified start-time
- **eSE_PriceRemovePriceEntry()** deletes the price entry with the specified start-time
- **eSE_PriceClearAllPriceEntries()** deletes all price entries in the list

These functions are fully detailed in Section 6.9.

6.5 Publishing Price Information

This section and its sub-sections describe the ways in which price information can be published (distributed) in an SE network. As introduced in Section 6.1, there are three ways in which price information may be published to the network from the Price cluster server (ESP):

- Unsolicited unicasts - refer to Section 6.5.1
- Response to a Get Current Price command - refer to Section 6.5.2
- Response to a Get Scheduled Prices command - refer to Section 6.5.3

All of the above methods require the ESP to send a Publish Price command to the relevant device(s), where the payload of this command includes information such as resource (e.g. gas), unit of measure, currency, price, current time, start-time and duration. On receipt of this command, if valid, the received price information will be automatically added to the price list on the device. If it is successfully added, an E_SE_PRICE_TABLE_ADD event will be generated on the receiving device and this event will be handled by the callback function registered for the relevant endpoint (see Section 4.3).

6.5.1 Unsolicited Price Updates

When the ESP receives updated price information from the utility company (via the backhaul network) or a new price tier becomes active, the ESP must inform all SE network devices that are using the Price cluster. The ESP therefore individually unicasts a Publish Price command to all these devices. This command is sent out automatically - there is no need for the application on the ESP to explicitly send the command. In the case of new prices received from the utility company, the ESP application must call the function **eSE_PriceAddPriceEntry()** to add the new price to the price list held by the server, and the Publish Price command is then automatically sent out (possibly with a 'start-time of now'). Note that if the stack has not been started when **eSE_PriceAddPriceEntry()** is called, the function's address mode parameter should be set to E_ZCL_AM_NO_TRANSMIT, so that no transmission is attempted.

It is recommended that price updates on the ESP are relayed to Price cluster clients with which the ESP has been (previously) bound.

Note: Each of these bindings is initiated on the client node (e.g. IPD) using the ZigBee PRO stack function **ZPS_eAplZdpBindUnbindRequest()** to add the client's address and endpoint to the Binding table on the ESP. Binding is described in the *ZigBee PRO Stack User Guide (JN-UG-3048)*.

Therefore, when updating its price list, the ESP application should call **eSE_PriceAddPriceEntry()** with the address mode parameter set to E_ZCL_AM_BOUND, so that the price updates are transmitted only to bound endpoints/nodes.

As an alternative to using binding, the ESP can maintain a list of SE network nodes that are able to receive unsolicited Publish Price commands at all times - that is, nodes with radio receivers that remain active during idle periods (e.g. when sleeping). Unsolicited updates are then only sent to clients in this group. The ESP gathers information for this group from the Get Current Price commands received from clients (see Section 6.5.2). This option requires the address node parameter to be set to ZPS_E_APL_AF_BROADCAST_RX_ON in **eSE_PriceAddPriceEntry()**.

The ESP can send unsolicited Publish Price commands with 'start-time of now' when an E_SE_PRICE_TABLE_ACTIVE event indicates that a new price has become active (see Section 6.8). This command can be used by devices that do not implement a real-time clock.

6.5.2 Get Current Price

Any device which supports the Price cluster can request the currently active price information from the ESP by sending a Get Current Price command. The SE API provides the function **eSE_PriceGetCurrentPriceSend()** which allows a Price cluster client to send this command to the Price cluster server and deal with the response.

- On receiving the command, the server automatically responds with a Publish Price command containing the requested price information.
- On receiving the response, the client checks whether the received price information is currently in the client's price list. If it is not, the client adds the new price information to the list and generates an E_SE_PRICE_TABLE_ADD event - this event is handled by the callback function registered for the relevant endpoint (see Section 4.3).

The Get Current Price command contains information on whether the radio receiver of the sending device remains active when the node is otherwise idle (e.g. sleeping). If this is true, the ESP application can use the address of the node to update a list of such devices, which it may use when sending out unsolicited Publish Price commands (see Section 6.5.1). The ESP application can extract this information from the event E_SE_PRICE_GET_CURRENT_PRICE_RECEIVED which is generated when a Get Current Price command is received by the server - this event is handled by the callback function registered for the relevant endpoint (see Section 4.3).

6.5.3 Get Scheduled Prices

Any device which supports the Price cluster can request the current price schedule from the ESP by sending a Get Scheduled Prices command - the schedule includes a set of prices with their start-times and durations. The SE API provides the function **eSE_PriceGetScheduledPricesSend()** which allows a Price cluster client to send this command to the Price cluster server and deal with the responses.

- On receiving the command, the server automatically responds with a sequence of Publish Price commands, where each of these responses contains the information for one scheduled price.
- On receiving a response, the client checks whether the received price information is currently in the client's price list. If it is not, the client adds the

new price information to the list and generates an E_SE_PRICE_TABLE_ADD event - this event is handled by the callback function registered for the relevant endpoint (see Section 4.3).

6.6 Time-Synchronisation via Publish Price Commands

As an alternative to using the Time cluster to time-synchronise an SE device with the ESP (as described in Section 6.3.3), the local application can use the time embedded in a Publish Price command from the ESP (see Section 6.5), as described below.

Note: A device which implements the Price cluster must also implement the Time cluster.

It is the responsibility of the application on an SE device to perform time-synchronisation with the ESP. This involves updating the ZCL time on the local device.

The initialisation of the ZCL time on a device should be performed using the Time cluster by requesting the current time from the ESP, as described in Section 6.3.2 (this method will also get time-zone and daylight saving information).

Subsequent re-synchronisations of a device with the time-master can use the time contained in Publish Price commands from the ESP (but note that no time-zone or daylight saving information is included). Therefore, a device can update its ZCL time whenever it receives a Publish Price command. On receiving this command, a 'data indication' stack event is generated, which causes JenOS to activate a ZCL user task. The event is initially handled by this task as described in Section 4.7, resulting in an E_ZCL_ZIGBEE_EVENT event being passed to the ZCL via **vZCL_EventHandler()**. The ZCL invokes the relevant user-defined callback function (see Section 4.7) which, provided that the event is of the type E_SE_PRICE_TIME_UPDATE, must update the ZCL time using **vZCL_SetUTCTime()**.

The demonstration application in the Application Note *Smart Energy HAN Solutions (JN-AN-1135)* illustrates how to do this.

Note that the `utctTime` field of the local copy of the Time cluster is not updated, since this should only be done following a read of the Time cluster attributes from the server.

***Caution:** If a device is handling Publish Price commands from more than one server, the time must only be updated with time events from one server, to prevent the time from jittering forwards and backwards if the servers' times are not in sync.*

The time-synchronisation of a device (with the time-master) should be performed regularly. As a rule, if no Publish Price commands have been received from the ESP in the last 48 hours, the device should request the current time from the ESP and update its own times as described in Section 6.3.3.

It is worth noting that an undefined ZCL time causes the following issues in the Price cluster (refer to Section 6.3.5 for information on checking ZCL time synchronisation):

- A Price cluster server without a ZCL time cannot issue any Publish Price commands, since the current time is a mandatory field of this command.
- A Price cluster client without a ZCL time cannot process a Publish Price command with a 'start-time of now', unless the ZCL time is first set with the time extracted from the received command.
- If the price at the head of the price list has a specified start-time, it is not possible to know whether this price is active or not.

Regarding the last point, a device should be time-synchronised with the ESP (as described in Section 6.3.2) before an attempt is made to add scheduled prices to the device's price list. Then, if the device receives a scheduled price with a 'start-time of now', it is permissible to add this price to the list.

6.7 Conversion Factor and Calorific Value (Gas Only)

The Price cluster provides attributes related to conversion factor and calorific value for use with gas (only):

- **Conversion factor:** Accounts for the variation of gas volume with temperature and pressure
- **Calorific value:** Indicates the quantity of energy in MJ that is generated per unit volume or unit mass of gas burned

The attributes associated with the above properties are part of the 'Commodity' set - refer to Section 6.2.

If required, conversion factor and/or calorific value must be enable in the compile-time options, as described in Section 6.13.

Conversion factors and calorific values can be independently scheduled with associated start-times. The Price cluster server (ESP) and clients each maintain a list of the scheduled conversion factors and a list of the scheduled calorific values (along with their start-times). The maximum number of entries in each list is by default 2 (allowing the present one and the next one to be stored), but this maximum can be re-defined in the compile-time options.

The ESP (Price cluster server) receives a scheduled conversion factor or calorific value from the utility company. A received value and its associated start-time are added as an entry to the relevant list on the server by the ESP application as follows:

- A new entry is added to the conversion factor list by calling the function **eSE_PriceAddConversionFactorEntry()**
- A new entry is added to the calorific value list by calling the function **eSE_PriceAddCalorificValueEntry()**

The entries are maintained in the list in increasing order of start-times. If an existing entry in the list has the same start-time as the new entry, the entry with the greater value of the Issuer Event ID is included in the list (and the other entry is discarded).

Once a new entry is added to a list on the server, a Publish Conversion Factor or Publish Calorific Value command is automatically sent to the cluster clients to inform them that a new value is available, allowing them to update their lists with the new information.

Initialising Conversion Factors and Calorific Values at Network Start-up

Note the following issues at network start-up:

- When the ESP node first starts, there may be no other active nodes in the network to receive a new conversion factor and/or calorific value. Thus, the Price cluster clients should request this information from the ESP when they start. They can do this using **eSE_PriceGetConversionFactorSend()** or **eSE_PriceGetCalorificValueSend()**, as appropriate.
- When initialising the conversion factor or calorific value at ESP start-up, the ESP application should call **eSE_PriceAddConversionFactorEntry()** or **eSE_PriceAddCalorificValueEntry()** with the address mode parameter set to E_ZCL_AM_NO_TRANSMIT. This prevents the new value from being transmitted to a network with no other active nodes.
- Any clients that are active during ESP initialisation should not request a conversion factor or calorific value from the ESP before the values are received from the utility company. To avoid this problem, the ESP application should obtain the values from the utility company before calling the ZigBee PRO function **ZPS_eAplZdoStartStack()** and after calling the relevant endpoint register function (e.g. **eSE_RegisterEspMeterEndPoint()**).

6.8 Price Events

The Price cluster has its own events that are handled through the callback mechanism outlined in Section 4.7 (and fully detailed in the *ZCL User Guide (JN-UG-3077)*). If a device uses the Price cluster then Price event handling must be included in the callback function for the associated endpoint, where this callback function is registered through the relevant endpoint registration function (for example, through **eSE_RegisterEspEndPoint()** for a standalone ESP). The relevant callback function will then be invoked when a Price event occurs.

For a Price event, the `eEventType` field of the `tsZCL_CallBackEvent` structure is set to E_ZCL_CBET_CLUSTER_CUSTOM. This event structure also contains an element `sClusterCustomMessage`, which is itself a structure containing a field `pvCustomData`. This field is a pointer to a `tsSE_PriceCallBackMessage` structure which contains the Price parameters:

```
typedef struct
{
    teSE_PriceCallBackEventType             eEventType;
    uint32                                  u32CurrentTime;

    union {
        tsSE_PriceTableCommand                  sPriceTableCommand;
        tsSE_PriceTableTimeEvent                sPriceTableTimeEvent;
        teSE_PriceCommandOptions                ePriceCommandOptions;
        tsSE_PriceAckCmdPayload                 *psAckCmdPayload;
        tsSE_PriceAttrReadInput                 sReadAttrInfo;
        tsSE_BlockPeriodTableTimeEvent          sBlockPeriodTableTimeEvent;
        tsSE_ConversionFactorTableTimeEvent     sConversionFactorTableTimeEvent;
        tsSE_CalorificValueTableTimeEvent       sCalorificValueTableTimeEvent;

    } uMessage;

} tsSE_PriceCallBackMessage;
```

The `eEventType` field of the above structure specifies the type of Price event that has been generated - these event types are listed and described below (also refer to Section 6.12.2 for a summary of the Price events).

Note: The field `sReadAttrInfo` is reserved for future use (for Block mode).

E_SE_PRICE_TABLE_ADD

The E_SE_PRICE_TABLE_ADD event is generated on a Price cluster client when an attempt has been made to add a scheduled price (received in a Publish Price command) to the local price list. In the `tsSE_PriceCallBackMessage` structure, the `u32CurrentTime` field is set to the current time from the Publish Price command and the `sPriceTableCommand` field is used as follows:

```
typedef struct {
    teSE_PriceStatus ePriceStatus;
} tsSE_PriceTableCommand;
```

`ePriceStatus` contains E_SE_PRICE_SUCCESS if a new price has been successfully added to the price list. Otherwise, the addition was rejected for the reason specified by `ePriceStatus`. If the addition was successful but the new price information overlapped (in time) any existing price information in the list, this previous price information may have been deleted from the list according to the rules in the ZigBee SE Profile specification.

E_SE_PRICE_TABLE_ACTIVE

The E_SE_PRICE_TABLE_ACTIVE event is generated when there is a new active price or the active price expires. This event can occur due to a time update or the reception of a Publish Price command from the server. In the `tsSE_PriceCallBackMessage` structure, the `u32CurrentTime` field is set to the current ZCL time and the `sPriceTableTimeEvent` field is used as follows:

```
typedef struct {
   teSE_PriceStatus  ePriceStatus;
   uint8             u8NumberOfEntriesFree;
} tsSE_PriceTableTimeEvent;
```

`ePriceStatus` contains E_SE_PRICE_SUCCESS if there is a new active price or E_SE_PRICE_TABLE_NOT_YET_ACTIVE if the price at the head of the list is scheduled for a time in the future.

`u8NumberOfEntriesFree` contains the number of free entries in the client's price list. This number can be used to determine whether the client should issue a new Get Scheduled Prices command, in order to obtain more price entries to fill the free space in the list.

E_SE_PRICE_GET_CURRENT_PRICE_RECEIVED

The E_SE_PRICE_GET_CURRENT_PRICE_RECEIVED event is generated on a Price cluster server when a Get Current Price command is received from a client. In the `tsSE_PriceCallBackMessage` structure, the `ePriceCommandOptions` field is used as follows:

```
typedef enum PACK
{
    E_SE_PRICE_REQUESTOR_RX_ON_IDLE = 0x01 // LSB set
} teSE_PriceCommandOptions;
```

This field indicates whether the client that sent the request has its radio receiver enabled when idle (e.g. sleeping), and is used as described in Section 6.5.1 and Section 6.5.2.

E_SE_PRICE_TIME_UPDATE

The E_SE_PRICE_TIME_UPDATE event is generated on a Price cluster client when a Publish Price command is received from the server. In the `tsSE_PriceCallBackMessage` structure, the `u32CurrentTime` field is set to the current time from the Publish Price command. The application may then use this information to time-synchronise the device, as described in Section 6.6.

E_SE_PRICE_ACK_RECEIVED

The E_SE_PRICE_ACK_RECEIVED event is generated on a Price cluster server when a Price Acknowledgment command is received from a client. In the `tsSE_PriceCallBackMessage` structure, the `psAckCmdPayload` field is a pointer to the structure `tsSE_PriceAckCmdPayload` defined as follows:

```
typedef struct {
    uint32              u32ProviderId;
    uint32              u32IssuerEventId;
    uint32              u32PriceAckTime;
    uint8               u8Control;
} tsSE_PriceAckCmdPayload;
```

This structure contains the Price Acknowledgement command payload.

E_SE_PRICE_NO_PRICE_TABLES

The E_SE_PRICE_NO_PRICE_TABLES event is generated when an active price expires, is deleted from the price list and the price list becomes empty. In the `tsSE_PriceCallBackMessage` structure the `sPriceTableTimeEvent` field is used as follows:

```
typedef struct {
   teSE_PriceStatus ePriceStatus;
   uint8 u8NumberOfEntriesFree;
} tsSE_PriceTableTimeEvent;
```

`ePriceStatus` contains E_SE_PRICE_NO_TABLES.

`u8NumberOfEntriesFree` contains the number of free entries in the client's price list. This number can be used to determine whether the client should issue a new Get Scheduled Prices command, in order to obtain more price entries to fill the free space in the list.

E_SE_PRICE_CONVERSION_FACTOR_TABLE_ACTIVE

The E_SE_PRICE_CONVERSION_FACTOR_TABLE_ACTIVE event is generated when a new conversion factor value becomes active - that is, when the start-time of the conversion factor entry becomes less than or equal to the present time. This event can occur due to a time update or the reception of a Publish Conversion Factor command from the server.

In the `tsSE_PriceCallBackMessage` structure, the `u32CurrentTime` field is set to the current ZCL time and the field `tsSE_PriceConversionFactorTableTimeEvent` is used as follows:

```
typedef struct {
    teSE_PriceStatus     eConversionFactorStatus;
    uint8                u8NumberOfEntriesFree;
} tsSE_ConversionFactorTableTimeEvent;
```

`eConversionFactorStatus` takes the value E_ZCL_SUCCESS when a new conversion factor becomes active.

`u8NumberOfEntriesFree` contains the present number of free entries in the conversion factor list. This value should be checked by the client before issuing a Get

Conversion Factor command to obtain a new conversion factor value - the command should be issued only if there is free space in the list for a new entry to be added.

E_SE_PRICE_CONVERSION_FACTOR_ADD

The E_SE_PRICE_CONVERSION_FACTOR_ADD event is generated when a new conversion factor entry is advertised by the ESP to the client application using the Publish Conversion Factor command. Note that the event is generated even when the new entry is not successfully added to the internal conversion factor list maintained by the cluster.

The status of the command is passed back to the user application in the `ePriceStatus` field of the `tsSE_PriceTableCommand` structure (see above) within the `tsSE_PriceCallBackMessage` structure.

E_SE_PRICE_CALORIFIC_VALUE_TABLE_ACTIVE

The E_SE_PRICE_CALORIFIC_VALUE_TABLE_ACTIVE event is generated when a new calorific value becomes active - that is, when the start-time of the calorific value entry becomes less than or equal to the present time. This event can occur due to a time update or the reception of a Publish Calorific Value command from the server.

In the `tsSE_PriceCallBackMessage` structure, the `u32CurrentTime` field is set to the current ZCL time and the field `tsSE_PriceCalorificValueTableTimeEvent` is used as follows:

```
typedef struct {
    teSE_PriceStatus        eCalorificValueStatus;
    uint8                   u8NumberOfEntriesFree;
} tsSE_CalorificValueTableTimeEvent;
```

`eCalorificValueStatus` takes the value E_ZCL_SUCCESS when a new calorific value becomes active.

`u8NumberOfEntriesFree` contains the present number of free entries in the calorific value list. This value should be checked by the client before issuing a Get Calorific Value command to obtain a new calorific value - the command should be issued only if there is free space in the list for a new entry to be added.

E_SE_PRICE_CALORIFIC_VALUE_ADD

The E_SE_PRICE_CALORIFIC_VALUE_ADD event is generated when a new calorific value entry is advertised by the ESP to the client application using the Publish Calorific Value command. Note that this event is generated even when the new entry is not successfully added to the internal calorific value list maintained by the cluster.

The status of the command is passed back to the user application in the `ePriceStatus` field of the `tsSE_PriceTableCommand` structure (see above) within the `tsSE_PriceCallBackMessage` structure.

6.9 Functions

The following Price cluster functions are provided in the SE API:

Function	**Page**
eSE_PriceCreate	97
eSE_PriceGetCurrentPriceSend	99
eSE_PriceGetScheduledPricesSend	100
eSE_PriceAddPriceEntry	102
eSE_PriceAddPriceEntryToClient	104
eSE_PriceGetPriceEntry	105
eSE_PriceDoesPriceEntryExist	106
eSE_PriceRemovePriceEntry	107
eSE_PriceClearAllPriceEntries	108
eSE_PriceAddConversionFactorEntry	109
eSE_PriceGetConversionFactorSend	111
eSE_PriceGetConversionFactorEntry	113
eSE_PriceDoesConversionFactorEntryExist	114
eSE_PriceRemoveConversionFactorEntry	115
eSE_PriceClearAllConversionFactorEntries	116
eSE_PriceAddCalorificValueEntry	117
eSE_PriceGetCalorificValueSend	119
eSE_PriceGetCalorificValueEntry	121
eSE_PriceDoesCalorificValueEntryExist	122
eSE_PriceRemoveCalorificValueEntry	123
eSE_PriceClearAllCalorificValueEntries	124

eSE_PriceCreate

teSE_PriceStatus eSE_PriceCreate(
bool_t *bIsServer*,
uint8 *u8NumberOfRecordEntries*,
uint8 **pu8AttributeControlBits*,
uint8 **pau8RateLabel*,
tsZCL_ClusterInstance **psClusterInstance*,
tsZCL_ClusterDefinition **psClusterDefinition*,
tsSE_PriceCustomDataStructure
**psCustomDataStructure*,
tsSE_PricePublishPriceRecord
**psPublishPriceRecord*,
void **pvEndPointSharedStructPtr***);**

Description

This function creates an instance of the Price cluster on an endpoint. The cluster instance is created on the endpoint which is associated with the supplied `tsZCL_ClusterInstance` structure and can act as a server or a client, as specified.

The function should only be called when setting up a custom endpoint containing one or more selected clusters (rather than the whole set of clusters supported by a standard ZigBee device). This function will create a Price cluster instance on the endpoint, but instances of other clusters may also be created on the same endpoint by calling their corresponding creation functions. For more details of creating cluster instances on custom endpoints, refer to Appendix B.

Note: This function must not be called for an endpoint on which a standard ZigBee device (e.g. IPD) will be used. In this case, the device and its supported clusters must be registered on the endpoint using the relevant device registration function from those described in Chapter 12.

When used, this function must be the first Price cluster function called in the application, and must be called after the stack has been started and after the application profile has been initialised.

The function requires an array to be declared for internal use, which contains one element (of type **uint8**) for each attribute of the cluster. The array length should therefore equate be the total number of attributes supported by the Price cluster, which can be obtained by using the macro CLD_PRICE_MAX_NUMBER_OF_ATTRIBUTE.

The array declaration should be as follows:

```
uint8
au8AppPriceClusterAttributeControlBits[CLD_PRICE_MAX_NUMBER_OF_ATTRIBUTE];
```

The function will initialise the array elements to zero.

The function also requires an array of price labels to be declared, in which each array element is a label (string) for each price in the price list. The required declarations are different for a cluster server and client, as follows:

```
uint8
au8RateLabel[SE_PRICE_NUMBER_OF_CLIENT_PRICE_RECORD_ENTRIES][SE_PRICE_CLIENT
_MAX_STRING_LENGTH];
```

```
uint8
au8RateLabel[SE_PRICE_NUMBER_OF_CLIENT_PRICE_RECORD_ENTRIES][SE_PRICE_CLIENT
_MAX_STRING_LENGTH];
```

Parameters

bIsServer	Type of cluster instance (server or client) to be created: TRUE - server FALSE - client
u8NumberOfRecordEntries	Number of prices that can be stored in the price list, one of: SE_PRICE_NUMBER_OF_SERVER_PRICE_RECORD_ENTRIES SE_PRICE_NUMBER_OF_CLIENT_PRICE_RECORD_ENTRIES
pu8AttributeControlBits	Pointer to an array of **uint8** values, with one element for each attribute in the cluster (see above).
pau8RateLabel	Pointer to an array of price labels (strings), with one element for each price in the price list (see above).
psClusterInstance	Pointer to structure containing information about the cluster instance to be created (see the *ZCL User Guide (JN-UG-3077)*). This structure will be updated by the function by initialising individual structure fields.
psClusterDefinition	Pointer to structure indicating the type of cluster to be created (see the *ZCL User Guide (JN-UG-3077)*). In this case, this structure must contain the details of the Price cluster. This parameter can refer to a pre-filled structure called `sCLD_Price` which is provided in the **Price.h** file.
psCustomDataStructure	Pointer to structure which contains custom data for the Price cluster. This structure is used for internal data storage. No knowledge of the fields of this structure is required
pvEndPointSharedStructPtr	Pointer to the shared structure used for attribute storage. This parameter should be the address of the structure of type `tsCLD_Price` which defines the attributes of Price cluster. The function will initialise the attributes with default values.

Returns

E_ZCL_SUCCESS

E_ZCL_FAIL

E_ZCL_ERR_PARAMETER_NULL

E_ZCL_ERR_INVALID_VALUE

eSE_PriceGetCurrentPriceSend

```
teZCL_Status eSE_PriceGetCurrentPriceSend(
        uint8 u8SourceEndPointId,
        uint8 u8DestinationEndPointId,
        tsZCL_Address *psDestinationAddress,
        uint8 *pu8TransactionSequenceNumber,
        teSE_PriceCommandOptions ePriceCommandOptions);
```

Description

This function can be used on a Price cluster client to send a Get Current Price command to the Price cluster server. Therefore, it is used by a device (such as an IPD) to obtain the currently active price from the ESP.

The ESP should respond with a Publish Price command containing the active price. This response is processed by the Price cluster. The obtained price is checked against the prices currently in the price list on the client. If the price is not currently in the list, it is added to the list and an E_SE_PRICE_TABLE_ADD event is generated to indicate that a price has been added.

A pointer must be specified to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response will be set to match the TSN in the request, allowing an incoming response to be paired with a request.

Parameters

u8SourceEndPointId	Number of the local endpoint through which the request will be sent
u8DestinationEndPointId	Number of the remote endpoint to which the request will be sent
psDestinationAddress	Pointer to a structure containing the address of the remote node to which the request will be sent
pu8TransactionSequenceNumber	Pointer to a location to store the Transaction Sequence Number (TSN) of the request
ePriceCommandOptions	Indicates whether the radio receiver on client remains on when the device is idle (e.g. asleep): 0x01 - receiver on when idle 0x00 - receiver off when idle An enumeration is provided for the 'on' case: E_SE_PRICE_REQUESTOR_RX_ON_IDLE

Returns

E_ZCL_SUCCESS

E_ZCL_FAIL

E_ZCL_ERR_PARAMETER_NULL

E_ZCL_ERR_EP_RANGE

E_ZCL_ERR_CLUSTER_NOT_FOUND

E_ZCL_ERR_ZBUFFER_FAIL

eSE_PriceGetScheduledPricesSend

```
teZCL_Status eSE_PriceGetScheduledPricesSend(
                        uint8 u8SourceEndPointId,
                        uint8 u8DestinationEndPointId,
                        tsZCL_Address *psDestinationAddress,
                        uint8 *pu8TransactionSequenceNumber,
                        uint32 u32StartTime,
                        uint8 u8NumberOfEvents);
```

Description

This function can be used on a Price cluster client to send a Get Scheduled Prices command to the Price cluster server. Therefore, it is used by a device (such as an IPD) to obtain the current price schedule from the ESP, either to check that its own price schedule is up-to-date or to recover the price schedule following a device reset.

You must specify the earliest start-time for the scheduled prices to be included in the results. This is normally set to zero or the current time (UTC). Note that you are not advised to specify the last time in the client price list, since the server may contain updates for prices covering an earlier time-period that are already in the client price list. You must also specify the maximum number of scheduled prices to be returned in the results.

The ESP should respond with multiple Publish Price commands containing the scheduled prices. Each response is processed by the Price cluster. The obtained price is checked against the prices currently in the price list on the client. If the price is not currently in the list, it is added to the list and an E_SE_PRICE_TABLE_ADD event is generated to indicate that a price has been added.

A pointer must be specified to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response will be set to match the TSN in the request, allowing an incoming response to be paired with a request.

Parameters

u8SourceEndPointId	Number of the local endpoint through which the request will be sent
u8DestinationEndPointId	Number of the remote endpoint to which the request will be sent
psDestinationAddress	Pointer to a structure containing the address of the remote node to which the request will be sent
pu8TransactionSequenceNumber	Pointer to a location to store the Transaction Sequence Number (TSN) of the request
u32StartTime	The earliest start-time of any prices to be returned - this is normally set to zero or the current time (UTC)
u8NumberOfEvents	The maximum number of scheduled prices to be returned in the results - this should normally be set to: SE_PRICE_NUMBER_OF_CLIENT_PRICE_RECORD_ENTRIES

Returns

E_ZCL_SUCCESS

E_ZCL_ERR_PARAMETER_NULL

E_ZCL_ERR_ZBUFFER_FAIL

eSE_PriceAddPriceEntry

```
teSE_PriceStatus eSE_PriceAddPriceEntry(
        uint8 u8SourceEndPointId,
        uint8 u8DestinationEndPointId,
        tsZCL_Address *psDestinationAddress,
        bool_t bOverwritePrevious,
        tsSE_PricePublishPriceCmdPayload *psPricePayload,
        uint8 *pu8TransactionSequenceNumber);
```

Description

This function can be used on the Price cluster server to add a price to the local price list. The function also sends an unsolicited Publish Price command containing the new price information to one or more remote endpoints. The function should be called on the ESP when a new price is received from the utility company.

On receiving the Publish Price command, a remote client will automatically add the new price to the local price list. However, you must specify the action to be taken if the time-period of the new price overlaps with the time-period of a price that is already in the client's price list. You can choose to delete the existing price and add the new price, or leave the existing price in place and not add the new price. The rules on overlapping prices are defined in the ZigBee Smart Energy Profile specification.

A pointer must be specified to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response will be set to match the TSN in the request, allowing an incoming response to be paired with a request.

Parameters

u8SourceEndPointId	Number of the local endpoint through which the request will be sent
u8DestinationEndPointId	Number of the remote endpoint to which the request will be sent
psDestinationAddress	Pointer to a structure containing the address of the remote node to which the Publish Price command will be sent. It is recommended that the command is sent to all bound clients using a ZCL address mode of E_ZCL_AM_BOUND. If the stack has not been started, the E_ZCL_AM_NO_TRANSMIT address mode should be used
bOverwritePrevious	Action to be taken if the new price overlaps (in time) a price which is already in the price list: TRUE - existing price deleted, new price added FALSE - new price not added and error returned
psPricePayload	Pointer to a structure containing the price information to be added (see Section 6.11.1). This parameter only needs to remain in scope for the duration of this function call

pu8TransactionSequenceNumber	Pointer to a location to store the Transaction Sequence Number (TSN) of the command

Returns

E_ZCL_SUCCESS
E_ZCL_FAIL
E_ZCL_ERR_PARAMETER_NULL
E_ZCL_ERR_INVALID_VALUE
E_ZCL_ERR_TIME_NOT_SYNCHRONISED
E_ZCL_ERR_INSUFFICIENT_SPACE
E_ZCL_ERR_EP_RANGE
E_ZCL_ERR_CLUSTER_NOT_FOUND
E_ZCL_ERR_ZBUFFER_FAIL
E_SE_PRICE_OVERFLOW
E_SE_PRICE_DUPLICATE
E_SE_PRICE_DATA_OLD

eSE_PriceAddPriceEntryToClient

```
teSE_PriceStatus eSE_PriceAddPriceEntryToClient(
    uint8 u8SourceEndPointId,
    bool_t bOverwritePrevious,
    tsSE_PricePublishPriceCmdPayload *psPricePayload);
```

Description

This function can be used on a Price cluster client to add a price to the local price list directly.

Normally, price entries are automatically added to the price list on a client when a Publish Price command is received from the server (e.g. the ESP). However, this function can be used by the local application to directly add a price entry to the price list on the client. The function should therefore only be used on a device which does not receive price information from the server (but by some other means, such as via the Internet).

Parameters

u8SourceEndPointId	Number of the local endpoint through which the request will be sent
bOverwritePrevious	Action to be taken if the new price overlaps (in time) a price which is already in the price list: TRUE - existing price deleted, new price added FALSE - new price not added and error returned
psPricePayload	Pointer to a structure containing the price information to be added (see Section 6.11.1). This parameter only needs to remain in scope for the duration of this function call

Returns

E_ZCL_SUCCESS
E_ZCL_FAIL
E_ZCL_ERR_PARAMETER_NULL
E_ZCL_ERR_INVALID_VALUE
E_ZCL_ERR_TIME_NOT_SYNCHRONISED
E_ZCL_ERR_INSUFFICIENT_SPACE
E_ZCL_ERR_EP_RANGE
E_ZCL_ERR_CLUSTER_NOT_FOUND
E_ZCL_ERR_ZBUFFER_FAIL
E_SE_PRICE_OVERFLOW
E_SE_PRICE_DUPLICATE
E_SE_PRICE_DATA_OLD

eSE_PriceGetPriceEntry

```
teSE_PriceStatus eSE_PriceGetPriceEntry(
        uint8 u8SourceEndPointId,
        bool_t bIsServer,
        uint8 u8TableIndex,
        tsSE_PricePublishPriceCmdPayload **psPricePayload);
```

Description

This function can be used to obtain the entry with specified index from a price list on the local device. For example, the function can be used on an IPD to obtain a price to display.

You must specify the endpoint on which the local Price cluster resides and whether this cluster instance is a server or a client.

Parameters

u8SourceEndPointId	Number of the local endpoint for the price list to be accessed
bIsServer	Nature of the Price cluster instance containing the price list: TRUE - server (e.g. on ESP) FALSE - client (e.g. on IPD)
u8TableIndex	The index of the price entry to obtain from the price list (index 0 is the entry with the oldest start-time and may contain the currently active price)
psPricePayload	Pointer to a pointer to a structure which will be used to store the obtained price information (see Section 6.11.1), if found. The pointer value that is returned in this parameter points to the structure in the internal storage associated with the list. The data in the structure will be valid as long as the item remains in the list

Returns

E_ZCL_SUCCESS

E_ZCL_FAIL

E_ZCL_ERR_PARAMETER_NULL

E_ZCL_ERR_CLUSTER_NOT_FOUND

E_SE_PRICE_TABLE_NOT_FOUND

eSE_PriceDoesPriceEntryExist

```
teSE_PriceStatus eSE_PriceDoesPriceEntryExist(
                        uint8 u8SourceEndPointId,
                        bool_t bIsServer,
                        uint32 u32StartTime);
```

Description

This function can be used to check whether a price entry with the specified start-time is present in a price list on the local device.

You must specify the endpoint on which the local Price cluster resides and whether this cluster instance is a server or a client.

For a price entry to be successfully found, the specified start-time must exactly match the start-time of an entry in the price list, otherwise the status code E_SE_PRICE_NOT_FOUND will be returned.

Parameters

u8SourceEndPointId	Number of the local endpoint for the price list to be accessed
bIsServer	Nature of the Price cluster instance containing the price list: TRUE - server FALSE - client
u32StartTime	Start-time of the price entry to search for

Returns

E_ZCL_SUCCESS

E_ZCL_ERR_CLUSTER_NOT_FOUND

E_SE_PRICE_NOT_FOUND

eSE_PriceRemovePriceEntry

```
teSE_PriceStatus eSE_PriceRemovePriceEntry(
                        uint8 u8SourceEndPointId,
                        bool_t bIsServer,
                        uint32 u32StartTime);
```

Description

This function can be used to delete a price entry with specified start-time from a price list on the local device.

You must specify the endpoint on which the local Price cluster resides and whether this cluster instance is a server or a client.

For the successful deletion of a price entry, the specified start-time must exactly match the start-time of an entry in the price list, otherwise the status code E_SE_PRICE_NOT_FOUND will be returned.

Parameters

u8SourceEndPointId	Number of the local endpoint on which Price cluster resides
bIsServer	Nature of the Price cluster instance containing the price list: TRUE - server FALSE - client
u32StartTime	The start-time of the price entry to delete

Returns

E_ZCL_SUCCESS

E_ZCL_ERR_CLUSTER_NOT_FOUND

E_SE_PRICE_NOT_FOUND

E_SE_PRICE_TABLE_NOT_FOUND

eSE_PriceClearAllPriceEntries

```
teSE_PriceStatus eSE_PriceClearAllPriceEntries(
                    uint8 u8SourceEndPointId,
                    bool_t bIsServer);
```

Description

This function can be used to delete all entries in a price list on the local device.

You must specify the endpoint on which the local Price cluster resides and whether this cluster instance is a server or a client.

Parameters

u8SourceEndPointId	Number of the local endpoint for the price list to be cleared
bIsServer	Nature of the Price cluster instance containing the price list: TRUE - server FALSE - client

Returns

E_ZCL_SUCCESS

E_ZCL_ERR_CLUSTER_NOT_FOUND

eSE_PriceAddConversionFactorEntry

```
teZCL_Status eSE_PriceAddConversionFactorEntry(
        uint8 u8SourceEndPointId,
        uint8 u8DestinationEndPointId,
        tsZCL_Address *psDestinationAddress,
        bool_t  bOverwritePrevious,
        tsSE_PricePublishConversionCmdPayload
                        *psPublishConversionCmdPayload,
        uint8 *pu8TransactionSequenceNumber);
```

Description

The function can be used on a Price cluster server to add a new conversion factor entry to the internal list of scheduled conversion factors maintained by the cluster. The function also sends an unsolicited Publish Conversion Factor command to the Price cluster client nodes in the network, to advertise the new conversion factor. Therefore, the function should be called on the ESP when a new conversion factor is received from the utility company.

On receiving the Publish Conversion Factor command, a remote client automatically adds the new conversion factor to the local conversion factor list. However, if the new entry has the same start-time as an existing entry in the list, the outcome depends on the setting of the boolean parameter *bOverwritePrevious* in this function:

- If this parameter is set to TRUE then the existing entry is removed and the new entry is added
- If this parameter is set to FALSE then the Issuer Event IDs of the two conversion factor entries are compared:
    - If the Event ID of the new entry is the greater, the existing entry is removed and the new entry is added
    - If the Event ID of the existing entry is the greater, E_ZCL_FAIL is returned and the list is not modified

A pointer must be specified to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response will be set to match the TSN in the request, allowing an incoming response to be paired with a request.

Parameters

u8SourceEndPointId	Number of the local endpoint through which the request will be sent
u8DestinationEndPointId	Number of the remote endpoint to which the request will be sent
psDestinationAddress	Pointer to a structure containing the address of the remote node to which the request will be sent

bOverwritePrevious	Determines whether an existing conversion factor with the same start-time on the clients will be over-written without comparing Event IDs (see above): TRUE - over-write existing entry FALSE - compare Event IDs first
psPublishConversionCmdPayload	Pointer to conversion factor entry to be added to list on server and advertised to clients
pu8TransactionSequenceNumber	Pointer to a location to store the Transaction Sequence Number (TSN) of the request

Returns

E_ZCL_SUCCESS

E_ZCL_ERR_PARAMETER_NULL

E_ZCL_ERR_TIME_NOT_SYNCHRONISED

eSE_PriceGetConversionFactorSend

```
teZCL_Status eSE_PriceGetConversionFactorSend(
                    uint8 u8SourceEndPointId,
                    uint8 u8DestinationEndPointId,
                    tsZCL_Address *psDestinationAddress,
                    uint8 *pu8TransactionSequenceNumber,
                    uint32 u32StartTime,
                    uint8 u8NumberOfEvents);
```

Description

The function can be used on a Price cluster client to send a Get Conversion Factor request to the Price cluster server. Therefore, it is used by a device (such as an IPD) to obtain scheduled conversion factor values from the ESP. The function allows scheduled conversion factors to be obtained with start-times greater than or equal to a specified time, *u32StartTime*.

The ESP should respond with a Publish Conversion Factor command containing up to *u8NumberOfEvent* scheduled conversion factor values. The Price cluster on the receiving client processes the response by updating the local conversion factor list, as follows. For each conversion factor received in the response, the event E_SE_PRICE_CONVERSION_FACTOR_ADD is generated.

A pointer must be specified to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response will be set to match the TSN in the request, allowing an incoming response to be paired with a request.

Parameters

u8SourceEndPointId	Number of the local endpoint through which the request will be sent
u8DestinationEndPointId	Number of the remote endpoint to which the request will be sent
psDestinationAddress	Pointer to a structure containing the address of the remote node to which the request will be sent
pu8TransactionSequenceNumber	Pointer to a location to store the Transaction Sequence Number (TSN) of the request
u32StartTime	Earliest start-time of scheduled conversion factors to be returned - a setting of 0 returns the factor that is currently active and factors with start-times in the future
u8NumberOfEvents	Maximum number of conversion factors to be returned as a result of this request

Returns

E_ZCL_SUCCESS
E_ZCL_ERR_PARAMETER_NULL
E_ZCL_ERR_EP_UNKNOWN
E_ZCL_ERR_ZBUFFER_FAIL
E_ZCL_ERR_EP_RANGE
E_ZCL_ERR_CLUSTER_NOT_FOUND
E_ZCL_ERR_ZTRANSMIT_FAIL

eSE_PriceGetConversionFactorEntry

```
teSE_PriceStatus eSE_PriceGetConversionFactorEntry(
        uint8  u8SourceEndPointId,
        bool_t  bIsServer,
        uint8 u8TableIndex,
        sSE_PricePublishConversionCmdPayload
                    **ppsPublishConversionCmdPayload);
```

Description

This function can be used to obtain the entry with the specified index from the conversion factor list on the local device. For example, the function can be used on an IPD to obtain a conversion factor to display.

You must specify the endpoint on which the local Price cluster resides and whether this cluster instance is a server or a client.

Parameters

u8SourceEndPointId	Number of the local endpoint for the conversion factor list to be accessed
bIsServer	Nature of the Price cluster instance containing the list: TRUE - server (e.g. on ESP) FALSE - client (e.g. on IPD)
u8TableIndex	The index of the entry to obtain from the conversion factor list (index 0 is the entry with the oldest start-time and may contain the currently active conversion factor)
***ppsPublishConversionCmdPayload*	Pointer to a pointer to a structure which will be used to store the obtained conversion factor information (see Section 6.11.2), if found. The pointer value that is returned in this parameter points to the structure in the internal storage associated with the list. The data in the structure will be valid as long as the item remains in the list

Returns

E_ZCL_SUCCESS

E_ZCL_FAIL

E_ZCL_ERR_PARAMETER_NULL

E_ZCL_ERR_CLUSTER_NOT_FOUND

E_SE_PRICE_TABLE_NOT_FOUND

eSE_PriceDoesConversionFactorEntryExist

```
teSE_PriceStatus
eSE_PriceDoesConversionFactorEntryExist(
                            uint8 u8SourceEndPointId,
                            bool_t bIsServer,
                            uint32 u32StartTime);
```

Description

This function can be used to check whether a conversion factor entry with the specified start-time is present in a conversion factor list on the local device.

You must specify the endpoint on which the local Price cluster resides and whether this cluster instance is a server or a client.

For a conversion factor entry to be successfully found, the specified start-time must exactly match the start-time of an entry in the conversion factor list, otherwise the status code E_SE_PRICE_NOT_FOUND will be returned.

Parameters

u8SourceEndPointId	Number of the local endpoint for the conversion factor list to be accessed
bIsServer	Nature of the Price cluster instance containing the price list: TRUE - server FALSE - client
u32StartTime	Start-time of the conversion factor entry to search for

Returns

E_ZCL_SUCCESS

E_ZCL_ERR_CLUSTER_NOT_FOUND

E_SE_PRICE_NOT_FOUND

eSE_PriceRemoveConversionFactorEntry

```
teSE_PriceStatus eSE_PriceRemoveConversionFactorEntry(
                            uint8 u8SourceEndPointId,
                            bool_t bIsServer,
                            uint32 u32StartTime);
```

Description

This function can be used to delete a conversion factor entry with specified start-time from conversion factor list on the local device.

You must specify the endpoint on which the local Price cluster resides and whether this cluster instance is a server or a client.

For the successful deletion of a conversion factor entry, the specified start-time must exactly match the start-time of an entry in the conversion factor list, otherwise the status code E_SE_PRICE_NOT_FOUND will be returned.

Parameters

u8SourceEndPointId	Number of the local endpoint for the conversion factor list to be accessed
bIsServer	Nature of the Price cluster instance containing the list: TRUE - server FALSE - client
u32StartTime	The start-time of the conversion factor entry to delete

Returns

E_ZCL_SUCCESS

E_ZCL_ERR_CLUSTER_NOT_FOUND

E_SE_PRICE_NOT_FOUND

E_SE_PRICE_TABLE_NOT_FOUND

eSE_PriceClearAllConversionFactorEntries

```
teSE_PriceStatus
eSE_PriceClearAllConversionFactorEntries(
                                      uint8 u8SourceEndPointId,
                                      bool_t bIsServer);
```

Description

This function can be used to delete all entries in a conversion factor list on the local device.

You must specify the endpoint on which the local Price cluster resides and whether this cluster instance is a server or a client.

Parameters

u8SourceEndPointId	Number of the local endpoint for the conversion factor list to be cleared
bIsServer	Nature of the Price cluster instance containing the price list: TRUE - server FALSE - client

Returns

E_ZCL_SUCCESS

E_ZCL_ERR_CLUSTER_NOT_FOUND

eSE_PriceAddCalorificValueEntry

```
teZCL_Status eSE_PriceAddCalorificValueEntry(
                    uint8 u8SourceEndPointId,
                    uint8 u8DestinationEndPointId,
                    tsZCL_Address *psDestinationAddress,
                    bool_t  bOverwritePrevious,
                    tsSE_PricePublishCalorificValueCmdPayload
                                *psPublishCalorificValueCmdPayload,
                    uint8 *pu8TransactionSequenceNumber);
```

Description

The function can be used on a Price cluster server to add a calorific value entry to the internal list of scheduled calorific values maintained by the cluster. The function also sends an unsolicited Publish Calorific Value command to the Price cluster client nodes in the network, to advertise the new calorific value. Therefore, the function should be called on the ESP when a new calorific value is received from the utility company.

On receiving the Publish Calorific Value command, a remote client automatically adds the new calorific value to the local calorific value list. However, if the new entry has the same start-time as an existing entry in the list, the outcome depends on the setting of the boolean parameter *bOverwritePrevious* in this function:

- If this parameter is set to TRUE then the existing entry is removed and the new entry is added
- If this parameter is set to FALSE then the Issuer Event IDs of the two calorific value entries are compared:
  - If the Event ID of the new entry is the greater, the existing entry is removed and the new entry is added
  - If the Event ID of the existing entry is the greater, E_ZCL_FAIL is returned and the list is not modified

A pointer must be specified to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response will be set to match the TSN in the request, allowing an incoming response to be paired with a request.

Parameters

u8SourceEndPointId	Number of the local endpoint through which the request will be sent
u8DestinationEndPointId	Number of the remote endpoint to which the request will be sent
psDestinationAddress	Pointer to a structure containing the address of the remote node to which the request will be sent

bOverwritePrevious	Determines whether an existing calorific value with the same start-time on the clients will be over-written without comparing Event IDs (see above): TRUE - over-write existing entry FALSE - compare Event IDs first
psPublishCalorificValueCmdPayload	Pointer to calorific value entry to be added to list on server and advertised to clients
pu8TransactionSequenceNumber	Pointer to a location to store the Transaction Sequence Number (TSN) of the request

Returns

E_ZCL_SUCCESS

E_ZCL_ERR_PARAMETER_NULL

E_ZCL_ERR_TIME_NOT_SYNCHRONISED

eSE_PriceGetCalorificValueSend

```
teZCL_Status eSE_PriceGetCalorificValueSend(
                        uint8 u8SourceEndPointId,
                        uint8 u8DestinationEndPointId,
                        tsZCL_Address *psDestinationAddress,
                        uint8 *pu8TransactionSequenceNumber,
                        uint32 u32StartTime,
                        uint8 u8NumberOfEvents);
```

Description

The function can be used on a Price cluster client to send a Get Calorific Value request to the Price cluster server. Therefore, it is used by a device (such as an IPD) to obtain scheduled calorific values from the ESP. The function allows scheduled calorific values to be obtained with start-times greater than or equal to a specified time, *u32StartTime*.

The ESP should respond with a Publish Calorific Value command containing up to *u8NumberOfEvent* scheduled calorific values. The Price cluster on the receiving client processes the response by updating the local calorific value list, as follows. For each calorific value received in the response, the event E_SE_PRICE_CALORIFIC_VALUE_ADD is generated.

A pointer must be specified to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response will be set to match the TSN in the request, allowing an incoming response to be paired with a request.

Parameters

u8SourceEndPointId	Number of the local endpoint through which the request will be sent
u8DestinationEndPointId	Number of the remote endpoint to which the request will be sent
psDestinationAddress	Pointer to a structure containing the address of the remote node to which the request will be sent
pu8TransactionSequenceNumber	Pointer to a location to store the Transaction Sequence Number (TSN) of the request
u32StartTime	Earliest start-time of scheduled calorific values to be returned - a setting of 0 returns the value that is currently active and values with start-times in the future
u8NumberOfEvents	Maximum number of calorific values to be returned as a result of this request

Returns

E_ZCL_SUCCESS
E_ZCL_ERR_PARAMETER_NULL
E_ZCL_ERR_EP_UNKNOWN
E_ZCL_ERR_ZBUFFER_FAIL
E_ZCL_ERR_EP_RANGE
E_ZCL_ERR_CLUSTER_NOT_FOUND
E_ZCL_ERR_ZTRANSMIT_FAIL

eSE_PriceGetCalorificValueEntry

```
teSE_PriceStatus eSE_PriceGetCalorificValueEntry(
        uint8  u8SourceEndPointId,
        bool_t  bIsServer,
        uint8 u8TableIndex,
        sSE_PricePublishCalorificValueCmdPayload
                        **ppsPublishCalorificValueCmdPayload);
```

Description

This function can be used to obtain the entry with the specified index from the calorific value list on the local device. For example, the function can be used on an IPD to obtain a calorific value to display.

You must specify the endpoint on which the local Price cluster resides and whether this cluster instance is a server or a client.

Parameters

u8SourceEndPointId	Number of the local endpoint for the calorific value list to be accessed
bIsServer	Nature of the Price cluster instance containing the list: TRUE - server (e.g. on ESP) FALSE - client (e.g. on IPD)
u8TableIndex	The index of the entry to obtain from the calorific value list (index 0 is the entry with the oldest start-time and may contain the currently active calorific value)
***ppsPublishCalorificValueCmdPayload*	Pointer to a pointer to a structure which will be used to store the obtained calorific value information (see Section 6.11.3), if found. The pointer value that is returned in this parameter points to the structure in the internal storage associated with the list. The data in the structure will be valid as long as the item remains in the list

Returns

E_ZCL_SUCCESS

E_ZCL_FAIL

E_ZCL_ERR_PARAMETER_NULL

E_ZCL_ERR_CLUSTER_NOT_FOUND

E_SE_PRICE_TABLE_NOT_FOUND

eSE_PriceDoesCalorificValueEntryExist

```
teSE_PriceStatus eSE_PriceDoesCalorificValueEntryExist(
                                 uint8 u8SourceEndPointId,
                                 bool_t bIsServer,
                                 uint32 u32StartTime);
```

Description

This function can be used to check whether a calorific value entry with the specified start-time is present in a calorific value list on the local device.

You must specify the endpoint on which the local Price cluster resides and whether this cluster instance is a server or a client.

For a calorific value entry to be successfully found, the specified start-time must exactly match the start-time of an entry in the calorific value list, otherwise the status code E_SE_PRICE_NOT_FOUND will be returned.

Parameters

u8SourceEndPointId	Number of the local endpoint for the calorific value list to be accessed
bIsServer	Nature of the Price cluster instance containing the price list: TRUE - server FALSE - client
u32StartTime	Start-time of the calorific value entry to search for

Returns

E_ZCL_SUCCESS

E_ZCL_ERR_CLUSTER_NOT_FOUND

E_SE_PRICE_NOT_FOUND

eSE_PriceRemoveCalorificValueEntry

```
teSE_PriceStatus eSE_PriceRemoveCalorificValueEntry(
                                    uint8 u8SourceEndPointId,
                                    bool_t bIsServer,
                                    uint32 u32StartTime);
```

Description

This function can be used to delete a calorific value entry with specified start-time from calorific value list on the local device.

You must specify the endpoint on which the local Price cluster resides and whether this cluster instance is a server or a client.

For the successful deletion of a calorific value entry, the specified start-time must exactly match the start-time of an entry in the calorific value list, otherwise the status code E_SE_PRICE_NOT_FOUND will be returned.

Parameters

u8SourceEndPointId	Number of the local endpoint for the calorific value list to be accessed
bIsServer	Nature of the Price cluster instance containing the list: TRUE - server FALSE - client
u32StartTime	Start-time of the calorific value entry to delete

Returns

E_ZCL_SUCCESS

E_ZCL_ERR_CLUSTER_NOT_FOUND

E_SE_PRICE_NOT_FOUND

E_SE_PRICE_TABLE_NOT_FOUND

eSE_PriceClearAllCalorificValueEntries

```
teSE_PriceStatus eSE_PriceClearAllCalorificValueEntries(
                        uint8 u8SourceEndPointId,
                        bool_t bIsServer);
```

Description

This function can be used to delete all entries in a calorific value list on the local device.

You must specify the endpoint on which the local Price cluster resides and whether this cluster instance is a server or a client.

Parameters

u8SourceEndPointId	Number of the local endpoint for the calorific value list to be cleared
bIsServer	Nature of the Price cluster instance containing the price list: TRUE - server FALSE - client

Returns

E_ZCL_SUCCESS

E_ZCL_ERR_CLUSTER_NOT_FOUND

6.10 Return Codes

In addition to some of the ZCL status enumerations (detailed in the *ZCL User Guide (JN-UG-3077)*), the following enumerations are returned by SE API Price cluster functions (see Section 6.9) to indicate the outcome of the function call.

```
typedef enum PACK
{
    E_SE_PRICE_OVERLAP =0x80,
    E_SE_PRICE_TABLE_NOT_YET_ACTIVE,
    E_SE_PRICE_DATA_OLD,
    E_SE_PRICE_NOT_FOUND,
    E_SE_PRICE_TABLE_NOT_FOUND,
    E_SE_PRICE_OVERFLOW,
    E_SE_PRICE_DUPLICATE,
    E_SE_PRICE_NO_TABLES,
    E_SE_PRICE_BLOCK_PERIOD_TABLE_NOT_YET_ACTIVE,
    E_SE_PRICE_NO_BLOCKS,
    E_SE_PRICE_NUMBER_OF_BLOCK_THRESHOLD_MISMATCH,
    E_SE_BLOCK_PERIOD_OVERFLOW,
    E_SE_BLOCK_PERIOD_DUPLICATE,
    E_SE_BLOCK_PERIOD_DATA_OLD,
    E_SE_BLOCK_PERIOD_OVERLAP,
    E_SE_PRICE_STATUS_ENUM_END
} teSE_PriceStatus;
```

The above enumerations are described in the table below.

Enumeration	Description
E_SE_PRICE_OVERLAP	New price overlaps (in time) with existing price in price list
E_SE_PRICE_TABLE_NOT_YET_ACTIVE	No active price at head of price list
E_SE_PRICE_DATA_OLD	Attempt made to add price which overlaps (in time) with existing price in price list and which is older than existing price *
E_SE_PRICE_NOT_FOUND	Specified price was not found in price list
E_SE_PRICE_TABLE_NOT_FOUND	Specified price list was not found
E_SE_PRICE_OVERFLOW	Attempt to add price to price list failed because end-time for new price (start-time + duration x 60) exceeds maximum permissible time value of 0xFFFFFFFF (UTC)
E_SE_PRICE_DUPLICATE	Specified price information already exists in price list
E_SE_PRICE_NO_TABLES	Reserved for future use (for Block mode)

Table 13: Price Cluster Return Codes

Enumeration	Description
E_SE_PRICE_BLOCK_PERIOD_TABLE_NOT_YET_ACTIVE	Reserved for future use (for Block mode)
E_SE_PRICE_NO_BLOCKS	Reserved for future use (for Block mode)
E_SE_PRICE_NUMBER_OF_BLOCK_THRESHOLD_MISMATCH	Reserved for future use (for Block mode)
E_SE_BLOCK_PERIOD_OVERFLOW	Reserved for future use (for Block mode)
E_SE_BLOCK_PERIOD_DUPLICATE	Reserved for future use (for Block mode)
E_SE_BLOCK_PERIOD_DATA_OLD	Reserved for future use (for Block mode)
E_SE_BLOCK_PERIOD_OVERLAP	Reserved for future use (for Block mode)

Table 13: Price Cluster Return Codes

* Value of `u32IssuerEventId` in `tsSE_PricePublishPriceCmdPayload` structure (see Section 6.11.1) is less for the price to be added than for the existing (overlapping) price.

6.11 Structures

6.11.1 tsSE_PricePublishPriceCmdPayload

This structure is used to hold price information to be added to a price list of a Price cluster:

```
typedef struct {
    uint8               u8UnitOfMeasure;
    uint8               u8PriceTrailingDigitAndPriceTier;
    uint8               u8NumberOfPriceTiersAndRegisterTiers;
    uint8               u8PriceRatio;
    uint8               u8GenerationPriceRatio;
    uint8               u8AlternateCostUnit;
    uint8               u8AlternateCostTrailingDigit;
    uint8               u8NumberOfBlockThresholds;
    uint8               u8PriceControl;
    uint16              u16Currency;
    uint16              u16DurationInMinutes;
    uint32              u32ProviderId;
    uint32              u32IssuerEventId;
    uint32              u32StartTime;
    uint32              u32Price;
    uint32              u32GenerationPrice;
    uint32              u32AlternateCostDelivered;
    tsZCL_OctetString   sRateLabel;

} tsSE_PricePublishPriceCmdPayload;
```

where:

- `u8UnitOfMeasure` indicates the resource (e.g. electricity) and unit of measure (e.g. kWh) for the pricing (see Section 8.10.3)
- `u8PriceTrailingDigitAndPriceTier` is an 8-bit bitmap indicating the price tier and the number of digits after the decimal point in the price:
    - The 4 most significant bits give the number of digits to the right of the decimal point in the price
    - The 4 least significant bits give the price tier in the range 1 to 6
- `u8NumberOfPriceTiersAndRegisterTiers` is an 8-bit bitmap indicating the number of price tiers available and the particular tier that the price information in the structure relates to:
    - The 4 most significant bits give the number of available price tiers in the range 0 to 6
    - The 4 least significant bits give the price tier used in the range 1 to 6 (this value must be less than or equal to the value in the 4 leading bits)
- `u8PriceRatio` (optional) is the ratio of the price quoted in `u32Price` to the 'normal' price offered by the utility company. The actual price ratio should be

multiplied by 10 for encoding this field, so that a field value of 0x01 represents 0.1 and 0xFE represents 25.4, while 0xFF indicates that the field is not used

- `u8GenerationPriceRatio` (optional) is the ratio of the price quoted in `u32GenerationPrice` to the 'normal' price offered by the utility company. The actual price ratio should be multiplied by 10 for encoding this field, so that a field value of 0x01 represents 0.1 and 0xFE represents 25.4, while 0xFF is reserved to indicate that the field is not used
- `u8AlternateCostUnit` (optional) is an 8-bit bitmap indicating the unit for the alternative cost in `u32AlternateCostDelivered`. Currently, the only supported unit is kilograms of CO_2, indicated by the value 0x01
- `u8AlternateCostTrailingDigit` (optional) is an 8-bit bitmap in which the 4 most significant bits indicate the number of digits after the decimal point in `u32AlternateCostDelivered` (the 4 least significant bits are reserved)
- `u8NumberOfBlockThresholds` is reserved for future use (for Block mode)
- `u8PriceControl` is reserved for future use (for Block mode)
- `u16Currency` indicates the currency (e.g. Euro) used for the price - this field should be set to the appropriate value defined by ISO 4217
- `u16DurationInMinutes` indicates the duration, in minutes, for which the price will be valid (0xFFFF indicates that price will remain valid until changed)
- `u32ProviderId` is an identifier for the utility company
- `u32IssuerEventId` is a unique identifier for the price information - the higher its value, the more recently the price information was issued (a UTC time-stamp could be used in this field)
- `u32StartTime` indicates the start-time (UTC) for the price, in seconds. The special value 0x00000000 denotes a start-time of 'now'
- `u32Price` is the resource price per unit indicated in `u8UnitOfMeasure`, expressed in the currency indicated in `u16Currency`, with the position of the decimal point as indicated in `u8PriceTrailingDigitAndPriceTier`
- `u32GenerationPrice` (optional) is the resource price per unit indicated in `u8UnitOfMeasure`, expressed in the currency indicated in `u16Currency` and with the position of the decimal point as indicated in `u8PriceTrailingDigitAndPriceTier`, for a resource that is generated on the customer premises and supplied to the utility company (e.g. solar-sourced electric power supplied to the national grid). A value of 0xFFFFFFFF indicates that this field is not used
- `u32AlternateCostDelivered` (optional) indicates an alternative cost (per resource consumption unit) which is measured by a means other than monetary - for example, the amount of CO_2 emitted per unit of gas consumed This alternative cost is interpreted as specified by `u8AlternateCostUnit` and `u8AlternateCostTrailingDigit`
- `sRateLabel` is a string of up to 12 characters containing a label for the price information in the structure

6.11.2 tsSE_PricePublishConversionCmdPayload

This structure is used to hold information to be added to a conversion factor list of a Price cluster:

```
typedef struct {
    uint32          u32IssuerEventId;
    uint32          u32StartTime;
    uint32          u32ConversionFactor;
    zbmap8          u8ConversionFactorTrailingDigit;
}tsSE_PricePublishConversionCmdPayload;
```

where:

- `u32IssuerEventId` is a unique identifier for the conversion factor information - the higher the value, the more recently the information was issued
- `u32StartTime` is the start-time of the conversion factor value. This is the time at which the conversion factor value is scheduled to become active
- `u32ConversionFactor` is used only for gas and accounts for the variation in the volume of gas with temperature and pressure (the value is dimensionless)
- `u8ConversionFactorTrailingDigit` is an 8-bit bitmap which indicates the location of the decimal point in the `u32ConversionFactor` field. The most significant 4 bits indicate the number of digits after the decimal point. The remaining bits are reserved

6.11.3 tsSE_PricePublishCalorificValueCmdPayload

This structure is used to hold information to be added to a calorific value list of the Price cluster:

```
typedef struct {
    zenum8                u8CalorificValueUnit;
    zbmap8                u8CalorificValueTrailingDigit;
    uint32                u32IssuerEventId;
    uint32                u32StartTime;
    uint32                u32CalorificValue;
}tsSE_PricePublishCalorificValueCmdPayload;
```

where:

- `u8CalorificValueUnit` is an 8-bit enumerated value which defines the unit for the `u32CalorificValue` field (below). It indicates whether the calorific value is quantified per unit volume or per unit mass - see Section 6.12.3.
- `u8CalorificValueTrailingDigit` is an 8-bit bitmap which indicates the location of the decimal point in the `u32CalorificValue` field (below). The most significant 4 bits indicate the number of digits after the decimal point. The remaining bits are reserved
- `u32IssuerEventId` is a unique identifier for the calorific value information - the higher the value, the more recently the information was issued
- `u32StartTime` is the start-time of the calorific value. This is the time at which the conversion factor value is scheduled to become active

- `u32CalorificValue` is used only for gas and indicates the quantity of energy in MJ that is generated per unit volume or unit mass of gas burned (see `u8CalorificValueUnit`). The position of the decimal point is indicated by `u8CalorificValueTrailingDigit` described above

6.12 Enumerations

6.12.1 'Attribute ID' Enumerations

The following structure contains the enumerations used to identify the attributes of the Price cluster.

Note: Only the Tier Label attributes are currently used. The remaining attributes are reserved for future use (for Block mode).

```
typedef enum PACK
{
    /* Price Cluster Attribute Tier Price Label Set Attr Ids
(D.4.2.2.1)*/
    E_CLD_P_ATTR_TIER_1_PRICE_LABEL = 0x0000,
    E_CLD_P_ATTR_TIER_2_PRICE_LABEL,
    :
    :
    E_CLD_P_ATTR_TIER_15_PRICE_LABEL,

    /* Price Cluster Attribute Block Threshold Set Attr IDs
(D.4.2.2.2)*/
    E_CLD_P_ATTR_BLOCK1_THRESHOLD = 0x0100,
    E_CLD_P_ATTR_BLOCK2_THRESHOLD,
    :
    :
    E_CLD_P_ATTR_BLOCK15_THRESHOLD,

    /* Price Cluster Attribute Block Period Set Attr IDs
(D.4.2.2.3)*/
    E_CLD_P_ATTR_START_OF_BLOCK_PERIOD = 0x0200,
    E_CLD_P_ATTR_BLOCK_PERIOD_DURATION,
    E_CLD_P_ATTR_THRESHOLD_MULTIPLIER,
    E_CLD_P_ATTR_THRESHOLD_DIVISOR,

    /* Price Cluster Attribute Commodity Set Attr IDs (D.4.2.2.4)*/
    E_CLD_P_ATTR_COMMODITY_TYPE = 0x0300,
    E_CLD_P_ATTR_STANDING_CHARGE,
    E_CLD_P_ATTR_CONVERSION_FACTOR,
    E_CLD_P_ATTR_CONVERSION_FACTOR_TRAILING_DIGIT,
    E_CLD_P_ATTR_CALORIFIC_VALUE,
    E_CLD_P_ATTR_CALORIFIC_VALUE_UNIT,
    E_CLD_P_ATTR_CALORIFIC_VALUE_TRAILING_DIGIT,

   /* Price Cluster Attribute Block Price Information Set Attr IDs
(D.4.2.2.5)*/
    E_CLD_P_ATTR_NOTIER_BLOCK1_PRICE = 0x0400,
    E_CLD_P_ATTR_NOTIER_BLOCK2_PRICE,
    :
    :
    E_CLD_P_ATTR_NOTIER_BLOCK16_PRICE,

    E_CLD_P_ATTR_TIER1_BLOCK1_PRICE = 0x0410,
    :
    E_CLD_P_ATTR_TIER1_BLOCK16_PRICE,

    E_CLD_P_ATTR_TIER2_BLOCK1_PRICE,
    :
    E_CLD_P_ATTR_TIER2_BLOCK16_PRICE,

    E_CLD_P_ATTR_TIER3_BLOCK1_PRICE,
    :
    E_CLD_P_ATTR_TIER3_BLOCK16_PRICE,

    E_CLD_P_ATTR_TIER4_BLOCK1_PRICE,
    :
    E_CLD_P_ATTR_TIER4_BLOCK16_PRICE,

    E_CLD_P_ATTR_TIER5_BLOCK1_PRICE,
    :
    E_CLD_P_ATTR_TIER5_BLOCK16_PRICE,

    E_CLD_P_ATTR_TIER6_BLOCK1_PRICE,
    :
    E_CLD_P_ATTR_TIER6_BLOCK16_PRICE,

    E_CLD_P_ATTR_TIER7_BLOCK1_PRICE,
    :
    E_CLD_P_ATTR_TIER7_BLOCK16_PRICE,
    E_CLD_P_ATTR_TIER8_BLOCK1_PRICE,
    :
    E_CLD_P_ATTR_TIER8_BLOCK16_PRICE,

    E_CLD_P_ATTR_TIER9_BLOCK1_PRICE,
    :
    E_CLD_P_ATTR_TIER9_BLOCK16_PRICE,

    E_CLD_P_ATTR_TIER10_BLOCK1_PRICE,
    :
    E_CLD_P_ATTR_TIER10_BLOCK16_PRICE,

    E_CLD_P_ATTR_TIER11_BLOCK1_PRICE,
    :
    E_CLD_P_ATTR_TIER11_BLOCK16_PRICE,

    E_CLD_P_ATTR_TIER12_BLOCK1_PRICE,
    :
    E_CLD_P_ATTR_TIER12_BLOCK16_PRICE,

    E_CLD_P_ATTR_TIER13_BLOCK1_PRICE,
    :
    E_CLD_P_ATTR_TIER13_BLOCK16_PRICE,

    E_CLD_P_ATTR_TIER14_BLOCK1_PRICE,
    :
    E_CLD_P_ATTR_TIER14_BLOCK16_PRICE,

    E_CLD_P_ATTR_TIER15_BLOCK1_PRICE,
    :
    E_CLD_P_ATTR_TIER15_BLOCK16_PRICE

    /* Price Cluster Billing Period Information Set Attr IDs */
    E_CLD_P_ATTR_START_OF_BILLING_PERIOD = 0x700,
    E_CLD_P_ATTR_BILLING_PERIOD_DURATION

} teCLD_SM_PriceAttributeID;
```

6.12.2 'Price Event' Enumerations

The event types generated by the Price cluster are enumerated in the `teSE_PriceCallBackEventType` structure below:

```
typedef enum PACK
{
    E_SE_PRICE_TABLE_ADD =0x00,
    E_SE_PRICE_TABLE_ACTIVE,
    E_SE_PRICE_GET_CURRENT_PRICE_RECEIVED,
    E_SE_PRICE_TIME_UPDATE,
    E_SE_PRICE_ACK_RECEIVED,
    E_SE_PRICE_NO_PRICE_TABLES,
    E_SE_PRICE_READ_BLOCK_PRICING,
    E_SE_PRICE_BLOCK_PERIOD_TABLE_ACTIVE,
    E_SE_PRICE_NO_BLOCK_PERIOD_TABLES,
    E_SE_PRICE_BLOCK_PERIOD_ADD,
    E_SE_PRICE_READ_BLOCK_THRESHOLDS,
    E_SE_PRICE_CONVERSION_FACTOR_TABLE_ACTIVE,
    E_SE_PRICE_CONVERSION_FACTOR_ADD,
    E_SE_PRICE_CALORIFIC_VALUE_TABLE_ACTIVE,
    E_SE_PRICE_CALORIFIC_VALUE_ADD,
    E_SE_PRICE_CBET_ENUM_END
} teSE_PriceCallBackEventType;
```

The above event types are described in Table 14 below.

Note: For further details on Price events, refer to Section 6.8.

Event Type Enumeration	Description
E_SE_PRICE_TABLE_ADD	Generated when a new scheduled price is added to the local price list
E_SE_PRICE_TABLE_ACTIVE	Generated when a new price becomes active or the active price expires
E_SE_PRICE_GET_CURRENT_PRICE_RECEIVED	Generated on the server when a Get Current Price command is received from a client
E_SE_PRICE_TIME_UPDATE	Generated on a client when a Publish Price command is received from the server
E_SE_PRICE_ACK_RECEIVED	Generated on a server when a Price Acknowledgment command is received from a client
E_SE_PRICE_NO_PRICE_TABLES	Generated when an active price expires, is deleted from the price list and the list becomes empty
E_SE_PRICE_READ_BLOCK_PRICING	Reserved for future use (for Block mode)
E_SE_PRICE_BLOCK_PERIOD_TABLE_ACTIVE	Reserved for future use (for Block mode)
E_SE_PRICE_NO_BLOCK_PERIOD_TABLES	Reserved for future use (for Block mode)
E_SE_PRICE_BLOCK_PERIOD_ADD	Reserved for future use (for Block mode)
E_SE_PRICE_READ_BLOCK_THRESHOLDS	Reserved for future use (for Block mode)
E_SE_PRICE_CONVERSION_FACTOR_TABLE_ACTIVE	Generated when a new conversion factor value becomes active
E_SE_PRICE_CONVERSION_FACTOR_ADD	Generated when a new conversion factor entry is advertised by the ESP via a Publish Conversion Factor command
E_SE_PRICE_CALORIFIC_VALUE_TABLE_ACTIVE	Generated when a new calorific value becomes active
E_SE_PRICE_CALORIFIC_VALUE_ADD	Generated when a new calorific value entry is advertised via a Publish Calorific Value command

Table 14: Price Event Types

6.12.3 'Calorific Value Unit' Enumerations

The possible units for the calorific value attribute of the Price cluster are enumerated in the `tsSE_PriceCalorificValueUnits` structure below:

```
typedef enum PACK
{
    E_SE_MEGA_JOULES_METER_CUBE  = 0x01,
    E_SE_MEGA_JOULES_KILOGRAM = 0x02
} tsSE_PriceCalorificValueUnits;
```

The above enumerations are described in Table 15 below.

Enumeration	Description
E_SE_MEGA_JOULES_METER_CUBE	Calorific value measured in MJ/m^3
E_SE_MEGA_JOULES_KILOGRAM	Calorific value measured in MJ/kg

Table 15: 'Calorific Value Unit' Enumerations

6.13 Compile-Time Options

This section describes the compile-time options that may be enabled in the **zcl_options.h** file of an application that uses the Price cluster.

The Price cluster is enabled by defining CLD_PRICE.

Client and server versions of the cluster are defined by PRICE_CLIENT and PRICE_SERVER, respectively.

Price List Size

The maximum number of prices that can be stored in the price list on a server and client defaults to five and two respectively. These default values can be over-ridden by assigning values to the corresponding macro below:

- SE_PRICE_NUMBER_OF_SERVER_PRICE_RECORD_ENTRIES
- SE_PRICE_NUMBER_OF_CLIENT_PRICE_RECORD_ENTRIES

Price Tier Label Attribute Set

The maximum number of supported Price Tier Label Attribute Sets can be defined by assigning a value between 1 and 15 (inclusive) to CLD_P_ATTR_TIER_PRICE_LABEL_MAX_COUNT.

Block Threshold Attribute Set

The maximum number of supported Block Threshold Attribute Sets can be defined by assigning a value between 1 to 15 (inclusive) to CLD_P_ATTR_BLOCK_THRESHOLD_MAX_COUNT.

Block Price Information Attribute Set

The maximum number of supported Block Price Information Attribute Sets can be defined by assigning a value (the maximum of which is shown below in brackets) to each of the following:

- CLD_P_ATTR_NO_TIER_BLOCK_PRICES_MAX_COUNT (16)
- CLD_P_ATTR_NUM_OF_TIERS_PRICE (15)
- CLD_P_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_PRICE (16)

Conversion Factor (Gas Only)

Conversion factor in the Price cluster is enabled by defining the macro PRICE_CONVERSION_FACTOR.

The attributes for conversion factor are enabled by defining the following macros:

- CLD_P_ATTR_CONVERSION_FACTOR
- CLD_P_ATTR_CONVERSION_FACTOR_TRAILING_DIGIT

The default value of the maximum number of entries that can be stored in the conversion factor list which is maintained on the Price cluster server and client is 2. This value can be over-ridden by assigning another value to the macro:

SE_PRICE_NUMBER_OF_CONVERSION_FACTOR_ENTRIES

Calorific Value (Gas Only)

Calorific value in the Price cluster is enabled by defining the macro PRICE_CALORIFIC_VALUE.

The attributes for calorific value are enabled by defining the following macros:

- CLD_P_ATTR_CALORIFIC_VALUE
- CLD_P_ATTR_CALORIFIC_VALUE_UNIT
- CLD_P_ATTR_CALORIFIC_VALUE_TRAILING_DIGIT

The default value of the maximum number of entries that can be stored in the calorific value list which is maintained on the server and client is 2. This value can be over-ridden by assigning another value to the macro:

SE_PRICE_NUMBER_OF_CALORIFIC_VALUE_ENTRIES

7. Messaging Cluster

This chapter outlines the Messaging cluster which is defined in the ZigBee Smart Energy profile. The cluster provides an interface for passing text messages from the utility company via the ESP to the other devices in a ZigBee SE network.

The Messaging cluster has a Cluster ID of 0x0703.

7.1 Overview

The Messaging cluster is required in SE devices as indicated in the table below.

	Server-side	**Client-side**
Mandatory in...	ESP	
Optional in...		Metering Device IPD PCT Smart Appliance

Table 16: Messaging Cluster in SE Devices

The ESP acts as the Messaging cluster server, since it is the device which receives the text messages from the utility company via the backhaul network. Other devices act as clients and receive the text messages forwarded by the ESP. Typically, the client is an IPD which displays user information messages from the utility company (e.g. new price information).

The Messaging cluster is enabled by defining CLD_MC in the **zcl_options.h** file - see Section 3.5.1. Further compile-time options for the Messaging cluster are detailed in Section 7.9. These options include the maximum size of the text string in a message, which is by default set to 80 characters.

Note: Large messages may be divided into fragments for transmission over the air. This fragmentation is automatically handled by the ZigBee PRO stack at the source and destination, but certain ZigBee network parameter values are required by the SE profile. These values are provided in Section 3.5.2.

7.2 Messaging Cluster Structure and Attributes

The Messaging cluster does not contain any attributes, only custom commands for passing messages between two ZigBee devices. The SE API provides functions for implementing these commands. These functions are referenced in Section 7.3 and are detailed in Section 7.5.

7.3 Message Delivery and Display

Text messages from the utility company, received by the ESP via the backhaul network, are normally forwarded (unsolicited) from the Messaging cluster server (ESP) to the cluster clients in the SE network. However, the clients can also request text messages that are stored on the ESP. These two scenarios are covered in the Section 7.3.2 and Section 7.3.3 respectively, but it is first necessary to understand how these messages are stored on the server and clients, as detailed in Section 7.3.1.

Note: Each text message has a start time and duration, and therefore an implied expiry time. The Messaging cluster will automatically delete a message once it has expired.

7.3.1 Storing Messages

A Messaging cluster server and client each maintains three lists of messages:

- **Active Message List:** This list contains (at most) only one message, the currently active message, at index 0.
- **Scheduled Message List:** This list contains a queue of scheduled messages, in start-time order with the next scheduled message at the head of the list.
- **Cancel Pending List:** This list is used to store messages that are pending cancellation (see event E_SE_MC_MESSAGE_CANCELLED in Section 7.4).

The total number of messages that can be stored across the three lists is defined in the **zcl_options.h** file using the following macros on the server and client respectively:

SE_MESSAGE_NUMBER_OF_SERVER_MESSAGE_RECORD_ENTRIES
SE_MESSAGE_NUMBER_OF_CLIENT_MESSAGE_RECORD_ENTRIES

By default, these are set to 2 messages.

Note that if a new message arrives with a ‘start-time of now’, it will over-write the currently active message. Also note that it is the responsibility of the application to obtain the currently active message from the Active Message List and display the message on the node - displaying a message is described in Section 7.3.4.

7.3.2 Forwarding a Message

It is the responsibility of the ESP application to receive and store text messages from the utility company and then to pass these messages to other devices in the SE network. A message is transmitted from the ESP to another SE device using the function **eSE_MCDisplayMessage()**. In order to avoid multiple calls to this function to pass a message to individual clients, you are advised to bind multiple clients to the server (ESP). Each of these bindings is initiated on the client node using the ZigBee PRO stack function **ZPS_eAplZdpBindUnbindRequest()** to add the client’s address and endpoint to the Binding table on the ESP. Alternatively, you can collect the clients into a group and specify its group address in the call to **eSE_MCDisplayMessage()**. Binding and group addressing are described in the *ZigBee PRO Stack User Guide (JN-UG-3048)*.

On receipt of the message at the client, the Messaging cluster automatically inserts the message into the client's Active Message List or Scheduled Message List, according to the start-time of the message.

When sending a message to a client, the server can request that a user confirmation is sent back by the client. A user confirmation requires some kind of user action on the client node to confirm that a displayed message has been read by the user - this user action is normally a button press (the node should also provide a visual indication to the user that a new message has been displayed and that a confirmation is required, e.g. a flashing light). This user action is not required until the message becomes active (and is displayed), which is indicated by an E_SE_MC_MESSAGE_ACTIVE event. On completion of the user action, the application must call the SE function **eSE_MCMessageConfirmationUserSend()** to send the confirmation to the server.

Note: For a message which is not immediately active, the confirmation will not be sent to the server until some time after the message was originally sent to the client - that is, until after the start-time of the message.

7.3.3 Requesting a Message

A Messaging cluster client can request from the server the last message received from the utility company, using the function **eSE_MCSendGetLastMessageRequest()**. The requested message may be displayed on the client once it has been received from the server. This function is useful if the client goes through a period in which it is unable to receive messages, particularly while sleeping - in this case, the function can be called on waking from sleep.

7.3.4 Displaying a Message

An SE device which implements the Messaging cluster as a client normally displays messages for user information purposes - the currently active message (in the Active Message List) is displayed. It is the responsibility of the application on the device to obtain this message from the list and display it.

The SE function **eSE_MCGetMessage()** allows a message to be extracted from a local message list on a Messaging cluster client - the relevant list must be specified and the required message is then specified by its index in the list. Thus, the application on a client can use this function to get the currently active message from the Active Message List in order to display the message. The application needs to do this each time a new message becomes active, which is indicated by the event E_SE_MC_MESSAGE_ACTIVE.

The application must also remove a message from the display when the message expires (indicated by the E_SE_MC_MESSAGE_EXPIRED event) or is cancelled (see Section 7.3.5).

7.3.5 Cancelling a Message

A previously sent message can be remotely cancelled on the client using the function **eSE_MCCancelMessage()** on the ESP. This function sends a request to the client to remove the specified message from the relevant message list on the client. Again, you are advised to use binding if the message is to be cancelled on all clients.

On receipt of the 'message cancel' request at the client, the event E_SE_MC_MESSAGE_CANCELLED is generated. The subsequent message cancellation process depends on whether a user confirmation request was included in the 'message cancel' request:

- If no user confirmation is required, the status field of the above event is E_SE_MC_SUCCESS and, in this case, the Messaging cluster automatically deletes the specified message from the Active Message List or Scheduled Message List.
- If a user confirmation is required, the status field of the above event is E_SE_MC_MESSAGE_CONFIRM_REQUIRED, in which case the following steps are required:
  a) The Messaging cluster moves the specified message from the Active Message List or Scheduled Message List to the Cancel Pending List.
  b) The application must first indicate on the display that a user action (e.g. a button press) is required in order to acknowledge the cancellation of the displayed message, and must then wait for this user input before removing the message from the display.
  c) The application must then call the function **eSE_MCMessageConfirmationUserSend()** to send the required confirmation to the server.
  d) The Messaging cluster automatically deletes the message from the Cancel Pending List.

7.4 Messaging Events

The Messaging cluster has its own events that are handled through the callback mechanism outlined in Section 4.7 (and fully detailed in the *ZCL User Guide (JN-UG-3077)*). If a device uses the Messaging cluster then Messaging event handling must be included in the callback function for the associated endpoint, e.g. for an IPD, this callback function is registered through **eSE_RegisterIPDEndPoint()**. The relevant callback function will then be invoked when a Messaging event occurs.

For a Messaging event, the `eEventType` field of the `tsZCL_CallBackEvent` structure is set to E_ZCL_CBET_CLUSTER_CUSTOM. This event structure also contains an element `sClusterCustomMessage`, which is itself a structure containing a field `pvCustomData`. This field is a pointer to a `tsSE_MCCallBackMessage` structure which contains the Messaging parameters:

```
typedef struct PACK
{
    teSE_MCCallBackEventType                    eEventType;
    uint8                                       u8CommandId;
    uint32                                      u32CurrentTime;
    teSE_MCStatus                               eMCStatus;

    union {
        tsSE_MCDisplayMessageCommandPayload     sDisplayMessageCommandPayload;
        tsSE_MCMessageConfirmCommandPayload     sMessageConfirmCommandPayload;
        tsSE_MCCancelMessageCommandPayload      sCancelMessageCommandPayload;
        // no get last message structure
    } uMessage;

} tsSE_MCCallBackMessage;
```

Information on the elements of the above structure is provided in the sub-sections below. The structure is fully detailed in Section 7.8.1.

7.4.1 Event Types

The `eEventType` field of the `tsSE_MCCallBackMessage` structure above specifies the type of Messaging event that has been generated - these event types are enumerated in the `teSE_MCCallBackEventType` structure and described below.

```
typedef enum PACK
{
     E_SE_MC_MESSAGE_COMMAND =0x00,
     E_SE_MC_MESSAGE_ACTIVE,
     E_SE_MC_MESSAGE_EXPIRED,
     E_SE_MC_MESSAGE_CANCELLED
} teSE_MCCallBackEventType;
```

E_SE_MC_MESSAGE_COMMAND

The E_SE_MC_MESSAGE_COMMAND event is generated when a command has been received on either the server or client. In the `tsSE_MCCallBackMessage` structure, the `u8CommandId` field is used to indicate the corresponding command - one of:

```
#define SE_MC_DISPLAY_MESSAGE  (0x00)
#define SE_MC_CANCEL_MESSAGE   (0x01)

#define SE_MC_GET_LAST_MESSAGE (0x00)
#define SE_MC_MESSAGE_CONFIRMATION (0x01)
```

E_SE_MC_MESSAGE_ACTIVE

The E_SE_MC_MESSAGE_ACTIVE event is generated when a new message has become active on either the client or server - that is, has moved to the Active Message List.

E_SE_MC_MESSAGE_EXPIRED

The E_SE_MC_MESSAGE_EXPIRED event is generated when a message has expired on either the client or server.

E_SE_MC_MESSAGE_CANCELLED

The E_SE_MC_MESSAGE_CANCELLED event is generated on a client when a message cancel request has been received from the server. The event has two different uses, depending on whether a user confirmation has been requested by the server (the confirmation request is included in the message cancel request):

- If no user confirmation is required, the Messaging cluster automatically deletes the specified message from the Active Message List or Scheduled Message List and the E_SE_MC_MESSAGE_CANCELLED event is generated.
- If a user confirmation is required, the Messaging cluster automatically sets the `eMCStatus` field of the `tsSE_MCCallBackMessage` structure to E_SE_MC_MESSAGE_CONFIRM_REQUIRED and then continues with the message cancellation as described in Section 7.3.5.

7.4.2 Other Elements of tsSE_MCCallBackMessage

In addition to the fields `eEventType` and `u8CommandId` described in Section 7.4.1, the `tsSE_MCCallBackMessage` structure contains the following elements.

u32CurrentTime

The `u32CurrentTime` field contains the time (UTC) at which the event was generated.

eMCStatus

The `eMCStatus` field indicates the status returned from the command that has been executed (the command identified in `u8CommandId`). The status codes are enumerated in the `teSE_MCStatus` structure, shown below.

```
typedef enum PACK
{
    E_SE_MC_SUCCESS =0x00,
    E_SE_MC_FAILURE,
    E_SE_MC_INVALID_VALUE,
    E_SE_MC_PARAMETER_NULL,
    E_SE_MC_INSUFFICIENT_SPACE,
    E_SE_MC_DUPLICATE_EXISTS,
    E_SE_MC_EP_NOT_FOUND,
    E_SE_MC_EP_RANGE,
    E_SE_MC_ZBUFFER_FAIL,
    E_SE_MC_MESSAGE_LATE,
    E_SE_MC_MESSAGE_NOT_FOUND,
    E_SE_MC_CLUSTER_NOT_FOUND,
    E_SE_MC_ZTRANSMIT_FAIL,
    E_SE_MC_TIME_NOT_SYNCHRONISED
} teSE_MCStatus;
```

uMessage

This field is a union of structures, containing a structure for each of the Messaging command payloads. The valid structure in the event is defined by the value of `u8CommandId`.

7.5 Functions

The following Messaging cluster functions are provided in the SE API:

Function	Page
eSE_MCCreate	145
eSE_MCDisplayMessage	147
eSE_MCCancelMessage	149
eSE_MCGetMessage	151
eSE_MCSendGetLastMessageRequest	152
eSE_MCMessageConfirmationUserSend	153

eSE_MCCreate

teZCL_Status eSE_MCCreate(
bool_t *bIsServer*,
uint8 *u8NumberOfMessageEntries*,
uint8 **pau8StringStorage*,
tsZCL_ClusterInstance **psClusterInstance*,
tsZCL_ClusterDefinition **psClusterDefinition*,
tsSE_MCCustomDataStructure
**psCustomDataStructure*,
tsSE_MCDisplayMessageCommandPayloadRecord
psDisplayMessageCommandPayloadRecord*);**

Description

This function creates an instance of the Messaging cluster on an endpoint.The cluster instance is created on the endpoint which is associated with the supplied `tsZCL_ClusterInstance` structure and can act as a server or a client, as specified.

The function should only be called when setting up a custom endpoint containing one or more selected clusters (rather than the whole set of clusters supported by a standard ZigBee device). This function will create a Messaging cluster instance on the endpoint, but instances of other clusters may also be created on the same endpoint by calling their corresponding creation functions. For more details of creating cluster instances on custom endpoints, refer to Appendix B.

Note: This function must not be called for an endpoint on which a standard ZigBee device (e.g. IPD) will be used. In this case, the device and its supported clusters must be registered on the endpoint using the relevant device registration function from those described in Chapter 12.

When used, this function must be the first Messaging cluster function called in the application, and must be called after the stack has been started and after the application profile has been initialised.

The function requires an array to be declared, in which each array element is a string corresponding to a stored message. The required declarations are different for a cluster server and client, as follows:

```
uint8
au8DisplayMessage[SE_MESSAGE_NUMBER_OF_SERVER_MESSAGE_RECORD_ENTRIES][SE_MES
SAGE_SERVER_MAX_STRING_LENGTH];
```

```
uint8
au8DisplayMessage[SE_MESSAGE_NUMBER_OF_CLIENT_MESSAGE_RECORD_ENTRIES][SE_MES
SAGE_SERVER_MAX_STRING_LENGTH];
```

Parameters

bIsServer	Type of cluster instance (server or client) to be created: TRUE - server FALSE - client
u8NumberOfRecordEntries	Number of messages that can be stored in the message lists, one of: SE_MESSAGE_NUMBER_OF_SERVER_MESSAGE_RECORD_ENTRIES SE_MESSAGE_NUMBER_OF_CLIENT_MESSAGE_RECORD_ENTRIES
pau8StringStorage	Pointer to an array of messages (strings), with one element for each stored message (see above).
psClusterInstance	Pointer to structure containing information about the cluster instance to be created (see the *ZCL User Guide (JN-UG-3077)*). This structure will be updated by the function by initialising individual structure fields.
psClusterDefinition	Pointer to structure indicating the type of cluster to be created (see the *ZCL User Guide (JN-UG-3077)*). In this case, this structure must contain the details of the Messaging cluster. This parameter can refer to a pre-filled structure called `sCLD_MC` which is provided in the **Messaging.h** file.
psCustomDataStructure	Pointer to structure which contains custom data for the Messaging cluster. This structure is used for internal data storage. No knowledge of the fields of this structure is required.
psDisplayMessageCommandPayloadRecord	Pointer to structure which will contain the payload of a message to be displayed.

Returns

E_ZCL_SUCCESS

E_ZCL_FAIL

E_ZCL_ERR_PARAMETER_NULL

E_ZCL_ERR_INVALID_VALUE

eSE_MCDisplayMessage

```
teSE_MCStatus eSE_MCDisplayMessage(
          uint8 u8SourceEndPointId,
          uint8 u8DestinationEndPointId,
          tsZCL_Address *psDestinationAddress,
          tsSE_MCDisplayMessageCommandPayload
                              *psDisplayMessageCommandPayload,
          uint8 *pu8TransactionSequenceNumber);
```

Description

This function can be used on the Messaging cluster server (normally the ESP) to send a request to a remote endpoint to display of the embedded message on the node. On receipt of the request, the client on the destination endpoint automatically inserts the message into the Active Message List or Scheduled Message List, according to the start-time of the message. The function also handles the deletion of the message once it has expired.

You must specify the address of the destination node and the destination endpoint number. It is possible to use this function to send a request to bound endpoints or to a group of endpoints on remote nodes - in the latter case, a group address must be specified. Note that when sending requests to multiple endpoints through a single call to this function, multiple responses will subsequently be received from the remote endpoints.

A user confirmation of the message can optionally be requested in the payload of the display request - this may require the customer to press a button to acknowledge that the message has been displayed. This confirmation is relayed to the cluster server using **eSE_MCMessageConfirmationUserSend()**. Refer to Section 7.3.2 for more information on user confirmations.

You are also required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response will be set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

Parameters

u8SourceEndPointId	Number of the local endpoint through which the request will be sent
u8DestinationEndPointId	Number of the remote endpoint to which the request will be sent. Note that this parameter is ignored when sending to address types E_ZCL_AM_BOUND and E_ZCL_AM_GROUP
psDestinationAddress	Pointer to a structure containing the address of the remote node to which the request will be sent
psDisplayMessageCommandPayload	Pointer to a structure (see Section 7.8.2) which holds the message to be displayed on the remote node

pu8TransactionSequenceNumber	Pointer to a location to store the Transaction Sequence Number (TSN) of the request

Returns

E_ZCL_SUCCESS
E_ZCL_FAIL
E_ZCL_ERR_PARAMETER_NULL
E_ZCL_ERR_INSUFFICIENT_SPACE
E_ZCL_ERR_ZBUFFER_FAIL
E_ZCL_ERR_ZTRANSMIT_FAIL
E_ZCL_ERR_CLUSTER_NOT_FOUND
E_SE_MC_MESSAGE_LATE

eSE_MCCancelMessage

```
teSE_MCStatus eSE_MCCancelMessage(
        uint8 u8SourceEndPointId,
        uint8 u8DestinationEndPointId,
        tsZCL_Address *psDestinationAddress,
        tsSE_MCCancelMessageCommandPayload
                        *psCancelMessageCommandPayload,
        uint8 *pu8TransactionSequenceNumber);
```

Description

This function can be used on the Messaging cluster server (normally the ESP) to request the cancellation of a message previously sent to a remote endpoint. This cancellation may involve removing the message from a display on the target device. A user confirmation of the message cancellation can optionally be requested in the payload of the request - that is, an action by the user (such as pressing a button) to acknowledge that the message has been withdrawn.

You must specify the address of the destination node and the destination endpoint number. It is possible to use this function to send a request to bound endpoints or to a group of endpoints on remote nodes - in the latter case, a group address must be specified. Note that when sending requests to multiple endpoints through a single call to this function, multiple responses will subsequently be received from the remote endpoints.

On receipt of the request, an E_SE_MC_MESSAGE_CANCELLED event is generated on the client. The subsequent actions depend on whether a user confirmation has been requested:

- If no user confirmation is required, the client on the destination endpoint automatically deletes the message from the Active Message List or Scheduled Message List, as appropriate, and removes the message from the display (if necessary).
- If a user confirmation is required, the client waits for the user action (e.g. button press). This client application must relay this confirmation to the cluster server using the function **eSE_MCMessageConfirmationUserSend()**. The message is then deleted.

Refer to Section 7.3.5 for more details of the message cancellation process.

You are also required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response will be set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

Parameters

u8SourceEndPointId	Number of the local endpoint through which the request will be sent
u8DestinationEndPointId	Number of the remote endpoint to which the request will be sent. Note that this parameter is ignored when sending to address types E_ZCL_AM_BOUND and E_ZCL_AM_GROUP

psDestinationAddress	Pointer to a structure containing the address of the remote node to which the request will be sent
psCancelMessageCommandPayload	Pointer to a structure (see Section 7.8.3) which holds details of the message to be cancelled (the message identifier and cancellation mechanism to be employed, including any user confirmation request)
pu8TransactionSequenceNumber	Pointer to a location to store the Transaction Sequence Number (TSN) of the request

Returns

E_ZCL_SUCCESS

E_ZCL_FAIL

E_ZCL_ERR_PARAMETER_NULL

E_ZCL_ERR_ZBUFFER_FAIL

E_ZCL_ERR_ZTRANSMIT_FAIL

E_ZCL_ERR_CLUSTER_NOT_FOUND

E_SE_MC_MESSAGE_LATE

E_SE_MC_MESSAGE_NOT_FOUND

eSE_MCGetMessage

```
teSE_MCStatus eSE_MCGetMessage(
          uint8 u8SourceEndPointId,
          uint8 u8tableIndex,
          teSE_MCEventList eEventList,
          tsSE_MCDisplayMessageCommandPayload
                        **ppsDisplayMessageCommandPayload);
```

Description

This function can be used to obtain a pointer to the entry with the specified index in the specified local message list (Active, Scheduled or Cancel Pending).

The function is normally used on an IPD to obtain the currently active message in order to display it - this should be done each time a new message becomes active, as indicated by an E_SE_MC_MESSAGE_ACTIVE event. Note that there is only one message in the Active Message List (therefore, for this list, only index 0 is valid).

Parameters

u8SourceEndPointId	Number of the local endpoint through which request will be made
u8tableIndex	Index of required message in the specified message list
eEventList	Message list to be queried (see Section 7.9), one of: E_SE_MC_MESSAGE_LIST_SCHEDULED E_SE_MC_MESSAGE_LIST_ACTIVE E_SE_MC_MESSAGE_LIST_CANCEL_PENDING
ppsDisplayMessageCommandPayload	Pointer to location to receive pointer to required entry in message list

Returns

E_ZCL_SUCCESS

E_ZCL_FAIL

E_ZCL_ERR_PARAMETER_NULL

E_SE_MC_MESSAGE_NOT_FOUND

eSE_MCSendGetLastMessageRequest

```
teSE_MCStatus eSE_MCSendGetLastMessageRequest(
                        uint8 u8SourceEndPointId,
                        uint8 u8DestinationEndPointId,
                        tsZCL_Address *psDestinationAddress,
                        uint8 *pu8TransactionSequenceNumber);
```

Description

This function can be used on a Messaging cluster client to request the last message sent by the server (normally the ESP). For example, this function may be called when a device wakes from sleep, in order to collect the last message missed while it was asleep.

You must specify the address of the destination node and the destination endpoint number.

You are also required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response will be set to match the TSN in the request, allowing an incoming response to be paired with a request.

Parameters

u8SourceEndPointId	Number of the local endpoint through which the request will be sent
u8DestinationEndPointId	Number of the remote endpoint to which the request will be sent
psDestinationAddress	Pointer to a structure containing the address of the remote node to which the request will be sent
pu8TransactionSequenceNumber	Pointer to a location to store the Transaction Sequence Number (TSN) of the request

Returns

E_ZCL_SUCCESS

E_ZCL_FAIL

E_ZCL_ERR_PARAMETER_NULL

E_ZCL_ERR_INSUFFICIENT_SPACE

E_ZCL_ERR_ZBUFFER_FAIL

E_ZCL_ERR_CLUSTER_NOT_FOUND

E_ZCL_ERR_ZTRANSMIT_FAIL

eSE_MCMessageConfirmationUserSend

```
teSE_MCStatus eSE_MCMessageConfirmationUserSend(
                        uint8 u8SourceEndPointId,
                        uint32 u32MessageId,
                        teSE_MCEventList eEventList);
```

Description

This function can be used on a Messaging cluster client to send a confirmation to the server that the user has acknowledged the display or cancellation of a message.

Thus, this function is required when a message display or message cancellation request has been received in which a user confirmation has been selected. The function should be called following a user action such as pressing a button. Refer to the Section 7.3 for more information on user confirmations.

Parameters

u8SourceEndPointId	Number of the local endpoint through which the confirmation will be sent
u32MessageId	Identifier of message for which user confirmation is required
eEventList	The message list which contains the message for which user confirmation is required. When confirming an active displayed message, the list will be E_SE_MC_MESSAGE_LIST_ACTIVE. When confirming a message cancellation, the list will be E_SE_MC_MESSAGE_LIST_CANCEL_PENDING

Returns

E_ZCL_SUCCESS

E_ZCL_FAIL

E_SE_MC_MESSAGE_NOT_FOUND

E_SE_MC_MESSAGE_CONFIRM_NOT_REQUIRED

7.6 Return Codes

In addition to some of the ZCL status enumerations (detailed in the *ZCL User Guide (JN-UG-3077)*), the following enumerations are returned by SE API Messaging cluster functions (see Section 7.5) to indicate the outcome of the function call.

```
typedef enum PACK
{
    E_SE_MC_DUPLICATE_EXISTS = 0x80,
    E_SE_MC_MESSAGE_LATE,
    E_SE_MC_MESSAGE_NOT_FOUND,
    E_SE_MC_MESSAGE_CONFIRM_REQUIRED,
    E_SE_MC_MESSAGE_CONFIRM_NOT_REQUIRED,
    E_SE_MC_STATUS_ENUM_END
} teSE_MCStatus;
```

The above enumerations are described in the table below.

Enumeration	**Description**
E_SE_MC_DUPLICATE_EXISTS	Message with specified identifier already exists in message lists
E_SE_MC_MESSAGE_LATE	Message end-time (start-time + duration x 60) is in the past
E_SE_MC_MESSAGE_NOT_FOUND	Message with specified identifier not found in message lists
E_SE_MC_MESSAGE_CONFIRM_ REQUIRED	User acknowledgement of message cancellation required
E_SE_MC_MESSAGE_CONFIRM_NOT_ REQUIRED	User acknowledgement of message cancellation not required

Table 17: Messaging Cluster Return Codes

7.7 Enumerations

7.7.1 'Message Event' Enumerations

The event types generated by the Messaging cluster are enumerated in the `teSE_MCCallBackEventType` structure below:

```
typedef enum PACK
{
    E_SE_MC_MESSAGE_COMMAND =0x00,
    E_SE_MC_MESSAGE_ACTIVE,
    E_SE_MC_MESSAGE_EXPIRED,
    E_SE_MC_MESSAGE_CANCELLED
} teSE_MCCallBackEventType;
```

The above event types are described in the table below.

Event Type Enumeration	Description
E_SE_MC_MESSAGE_COMMAND	Generated on a cluster client when a command is received from the server
E_SE_MC_MESSAGE_ACTIVE	Indicates that there is a new active message to display
E_SE_MC_MESSAGE_EXPIRED	Indicates that the currently active/displayed message has expired (reached its end-time)
E_SE_MC_MESSAGE_CANCELLED	Indicates that a message has been deleted from the message lists

Table 18: Messaging Event Types

7.7.2 'Message List' Enumerations

This structure contains enumerations used to identify a message list (Active Message List, Scheduled Message List and Cancel Pending List):

```
typedef enum PACK
{
    E_SE_MC_MESSAGE_LIST_SCHEDULED = 0x00,
    E_SE_MC_MESSAGE_LIST_ACTIVE,
    E_SE_MC_MESSAGE_LIST_DEALLOCATED,
    E_SE_MC_MESSAGE_LIST_CANCEL_PENDING,
    E_SE_MC_MESSAGE_LIST_ENUM_END
} teSE_MCEventList;
```

The above enumerations are described in the table below.

Enumeration	Description
E_SE_MC_MESSAGE_LIST_SCHEDULED	Refers to Scheduled Message List, which holds messages that are not yet active (i.e. the message start time is later than the current time)
E_SE_MC_MESSAGE_LIST_ACTIVE	Refers to Active Message List, which holds the currently active message, if there is one (i.e. a message with start time before the current time and start time+duration after the current time)
E_SE_MC_MESSAGE_LIST_DEALLOCATED	Used internally by the cluster
E_SE_MC_MESSAGE_LIST_CANCEL_PENDING	Refers to Cancel Pending List, which holds a message for which a cancel confirmation has been requested but not sent - see eSE_MCMessageConfirmationUserSend on page 153

Table 19: Message List Enumerations

7.8 Structures

7.8.1 tsSE_MCCallBackMessage

For a Messaging event, the `eEventType` field of the `tsZCL_CallBackEvent` structure is set to E_ZCL_CBET_CLUSTER_CUSTOM. This event structure also contains an element `sClusterCustomMessage`, which is itself a structure containing a field `pvCustomData`. This field is a pointer to the following `tsSE_MCCallBackMessage` structure which contains the Messaging parameters:

```
typedef struct PACK
{
    teSE_MCCallBackEventType                          eEventType;
    uint8                                             u8CommandId;
    uint32                                            u32CurrentTime;
    teSE_MCStatus                                     eMCStatus;

    union {
        tsSE_MCDisplayMessageCommandPayload           sDisplayMessageCommandPayload;
        tsSE_MCMessageConfirmCommandPayload           sMessageConfirmCommandPayload;
        tsSE_MCCancelMessageCommandPayload            sCancelMessageCommandPayload;
        // no get last message structure
    } uMessage;

} tsSE_MCCallBackMessage;
```

where:

- `eEventType` is the messaging event type from those listed in Section 7.7
- `u8CommandId` is the identifier of the type of messaging command received. This field is only valid for the messaging event type E_SE_MC_MESSAGE_COMMAND and, when valid, takes one of the following enumerated values:

  SE_MC_DISPLAY_MESSAGE (0x00, server to client)
  SE_MC_CANCEL_MESSAGE (0x01, server to client)
  SE_MC_GET_LAST_MESSAGE (0x00, client to server)
  SE_MC_MESSAGE_CONFIRMATION (0x01, client to server)
- `u32CurrentTime` is the current time (UTC), in seconds
- `eMCStatus` is the status returned after executing the command specified in the field `u8CommandId`. The possible status values are listed and described in Section 7.6
- `uMessage` is a union containing the command payload in one of the following forms (depending on the command specified in the field `u8CommandId`):
  - `sDisplayMessageCommandPayload` is a structure containing the payload of a 'display message' command - see Section 7.8.2
  - `sMessageConfirmCommandPayload` is a structure containing the payload of a 'confirm' command - see Section 7.8.4
  - `sCancelMessageCommandPayload` is a structure containing the payload of a 'cancel message' command - see Section 7.8.3

7.8.2 tsSE_MCDisplayMessageCommandPayload

This structure is used to hold the payload of ‘display message’ command sent from the Messaging cluster server to a client:

```
typedef struct {
    uint8                        u8MessageControl;
    uint16                       u16DurationInMinutes;
    uint32                       u32MessageId;
    uint32                       u32StartTime;
    tsZCL_CharacterString        sMessage;
} tsSE_MCDisplayMessageCommandPayload;
```

where:

- `u8MessageControl` is an 8-bit bitmap, where:
  - Bits 1-0 are used to specify the transmission options of the command
  - Bits 3-2 are used to specify the priority of the command
  - Bits 6-4 are reserved
  - Bit 7 is used to indicate whether a user confirmation is required

  For further details, refer to the *ZigBee Smart Energy Profile Specification*.
- `u16DurationInMinutes` is the duration of validity for the message, in minutes
- `u32MessageId` is the unique identifier of the embedded message
- `u32StartTime` is the start-time (UTC) for the message, in seconds
- `sMessage` is the character string containing the message (this string is, by default, up to 80 characters long but this maximum can be configured in the **zcl_options.h** header file)

7.8.3 tsSE_MCCancelMessageCommandPayload

This structure is used to hold the payload of 'cancel message' command sent from the Messaging cluster server to a client:

```
typedef struct PACK {
    uint32                      u32MessageId;
    uint8                       u8MessageControl;
} tsSE_MCCancelMessageCommandPayload;
```

where:

- `u32MessageId` is the unique identifier of the message to be cancelled
- `u8MessageControl` is an 8-bit bitmap, where:
    - Bits 1-0 are used to specify the transmission options of the command
    - Bits 3-2 are used to specify the priority of the command
    - Bits 6-4 are reserved
    - Bit 7 is used to indicate whether a user confirmation is required

  For further details, refer to the *ZigBee Smart Energy Profile Specification*.

7.8.4 tsSE_MCMessageConfirmCommandPayload

This structure is used to hold the payload of 'confirmation' command sent from a Messaging cluster client to the server:

```
typedef struct PACK {
    uint32                      u32MessageId;
    uint32                      u32ConfirmationTime;
} tsSE_MCMessageConfirmCommandPayload;
```

where:

- `u32MessageId` is the unique identifier of the message being confirmed
- `u32ConfirmationTime` is the time (UTC) at which the message was confirmed, in seconds

7.9 Compile-Time Options

This section describes the compile-time options that may be enabled in the **zcl_options.h** file of an application that uses the Messaging cluster.

The Messaging cluster is enabled by defining CLD_MC.

Client and server versions of the cluster are defined by MC_CLIENT and MC_SERVER, respectively.

Message List Size

The maximum number of messages that can be stored in the message lists on a server and on a client defaults to two. This number can be over-ridden for a server and client, respectively, by assigning values to:

- SE_MESSAGE_NUMBER_OF_SERVER_MESSAGE_RECORD_ENTRIES
- SE_MESSAGE_NUMBER_OF_CLIENT_MESSAGE_RECORD_ENTRIES

Message Length

The following macro can be used to set the length (in characters) of each message:

SE_MESSAGE_SERVER_MAX_STRING_LENGTH

For example, to set the message length to 80 characters:

```
#define SE_MESSAGE_SERVER_MAX_STRING_LENGTH  (80)
```

Note that for a string-length of 80 characters (as set by the above macro), the ZigBee PRO stack must be able to handle fragmented APDUs for message transmission.

8. Simple Metering Cluster

This chapter outlines the Simple Metering cluster which is defined in the ZigBee Smart Energy profile and is used to handle information relating to the measured consumption of some resource, which may be electricity, gas, heat or water.

The Simple Metering cluster has a Cluster ID of 0x0702.

8.1 Overview

The Simple Metering cluster is required in SE devices as indicated in the table below.

	Server-side	**Client-side**
Mandatory in...	Metering Device	
Optional in...	ESP	ESP IPD PCT

Table 20: Simple Metering Cluster in SE Devices

Thus, a Metering Device or ESP can use this cluster to store attributes and respond to commands relating to these attributes, while an IPD or PCT may use this cluster to issue commands to interact with remote attributes held on a Metering Device or ESP.

The Simple Metering cluster is enabled by defining CLD_SIMPLE_METERING in the **zcl_options.h** file - see Section 3.5.1. Further compile-time options for the Simple Metering cluster are detailed in Section 8.12.

The information that can potentially be stored in this cluster is organised into the following attribute sets:

- Reading Information Set (resource measurement information)
- TOU Information Set (Time-Of-Use information)
- Meter Status
- Formatting (data formatting/interpretation guidance)
- Historical Consumption
- Load Profile Configuration
- Supply Limit
- Block Information (for future use - not certifiable in SE 1.1.1 or earlier)
- Alarms (for future use - not certifiable in SE 1.1.1 or earlier)

This information is stored in both mandatory and optional attributes - see Section 8.3.

Note: Many of the Simple Metering cluster attributes are not certifiable in SE 1.1.1 (07-5356-17) or earlier and are reserved for future use (as indicated in Section 8.2).

8.2 Simple Metering Cluster Structure and Attributes

The Simple Metering cluster is contained in the following `tsSE_SimpleMetering` structure:

```
typedef struct
{
    /* Reading information attribute set attribute ID's (D.3.2.2.1) */
    zuint48           u48CurrentSummationDelivered;        /* Mandatory */

#ifdef CLD_SM_ATTR_CURRENT_SUMMATION_RECEIVED
    zuint48           u48CurrentSummationReceived;
#endif

#ifdef CLD_SM_ATTR_CURRENT_MAX_DEMAND_DELIVERED
    zuint48           u48CurrentMaxDemandDelivered;
#endif

#ifdef CLD_SM_ATTR_CURRENT_MAX_DEMAND_RECEIVED
    zuint48           u48CurrentMaxDemandReceived;
#endif

#ifdef CLD_SM_ATTR_DFT_SUMMATION
    zuint48           u48DFTSummation;
#endif

#ifdef CLD_SM_ATTR_DAILY_FREEZE_TIME
    zuint16           u16DailyFreezeTime;
#endif

#ifdef CLD_SM_ATTR_POWER_FACTOR
    zint8             i8PowerFactor;
#endif

#ifdef CLD_SM_ATTR_READING_SNAPSHOT_TIME
    zutctime          utctReadingSnapshotTime;
#endif

#ifdef CLD_SM_ATTR_CURRENT_MAX_DEMAND_DELIVERED_TIME
    zutctime          utctCurrentMaxDemandDeliveredTime;
#endif

#ifdef CLD_SM_ATTR_CURRENT_MAX_DEMAND_RECEIVED_TIME
    zutctime          utctCurrentMaxDemandReceivedTime;
#endif

#ifdef CLD_SM_ATTR_DEFAULT_UPDATE_PERIOD
    uint8             u8DefaultUpdatePeriod;
#endif

#ifdef CLD_SM_ATTR_FAST_POLL_UPDATE_PERIOD
    uint8             u8FastPollUpdatePeriod;
#endif
```

```
#ifdef CLD_SM_ATTR_CURRENT_BLOCK_PERIOD_CONSUMPTION_DELIVERED
    zuint48           u48CurrentBlockPeriodConsumptionDelivered;
#endif

#ifdef CLD_SM_ATTR_DAILY_CONSUMPTION_TARGET
    zuint24           u24DailyConsumptionTarget;
#endif

#ifdef CLD_SM_ATTR_CURRENT_BLOCK
    zenum8            e8CurrentBlock;
#endif

#ifdef CLD_SM_SUPPORT_GET_PROFILE

#ifdef CLD_SM_ATTR_PROFILE_INTERVAL_PERIOD
    zenum8            eProfileIntervalPeriod;
#endif

#ifdef CLD_SM_ATTR_INTERVAL_READ_REPORTING_PERIOD
    uint16            u16IntervalReadReportingPeriod;
#endif

#endif  // CLD_SM_SUPPORT_GET_PROFILE

#ifdef CLD_SM_ATTR_PREVIOUS_BLOCK_PERIOD_CONSUMPTION_DELIVERED
    zuint48           u48PreviousBlockPeriodConsumptionDelivered;
#endif

#ifdef CLD_SM_ATTR_PRESET_READING_TIME
    uint16            u16PresetReadingTime;
#endif

#ifdef CLD_SM_ATTR_VOLUME_PER_REPORT
    uint16            u16VolumePerReport;
#endif

#ifdef CLD_SM_ATTR_FLOW_RESTRICTION
    uint8             u8FlowRestriction;
#endif

#ifdef CLD_SM_ATTR_SUPPLY_STATUS
    zbmap8            u8SupplyStatus;
#endif

#ifdef CLD_SM_ATTR_CURRENT_INLET_ENERGY_CARRIER_SUMMATION
    zuint48           u48CurrentInletEnergyCarrierSummation;
#endif

#ifdef CLD_SM_ATTR_CURRENT_OUTLET_ENERGY_CARRIER_SUMMATION
    zuint48           u48CurrentOutletEnergyCarrierSummation;
#endif

#ifdef CLD_SM_ATTR_INLET_TEMPERATURE
```

```
        int16             i16InletTemperature;
    #endif

    #ifdef CLD_SM_ATTR_OUTLET_TEMPERATURE
        int16             i16OutletTemperature;
    #endif

    #ifdef CLD_SM_ATTR_CONTROL_TEMPERATURE
        int16             i16ControlTemperature;
    #endif

    #ifdef CLD_SM_ATTR_CURRENT_INLET_ENERGY_CARRIER_DEMAND
        zint24            i24CurrentInletEnergyCarrierDemand;
    #endif

    #ifdef CLD_SM_ATTR_CURRENT_OUTLET_ENERGY_CARRIER_DEMAND
        zint24            i24CurrentOutletEnergyCarrierDemand;
    #endif

        /* Time Of Use Information attribute attribute ID's set (D.3.2.2.2) */
    #ifdef CLD_SM_ATTR_CURRENT_TIER_1_SUMMATION_DELIVERED
        zuint48           u48CurrentTier1SummationDelivered;
    #endif

    #ifdef CLD_SM_ATTR_CURRENT_TIER_1_SUMMATION_RECEIVED
        zuint48           u48CurrentTier1SummationReceived;
    #endif

    #ifdef CLD_SM_ATTR_CURRENT_TIER_2_SUMMATION_DELIVERED
        zuint48           u48CurrentTier2SummationDelivered;
    #endif

    #ifdef CLD_SM_ATTR_CURRENT_TIER_2_SUMMATION_RECEIVED
        zuint48           u48CurrentTier2SummationReceived;
    #endif

    #ifdef CLD_SM_ATTR_CURRENT_TIER_3_SUMMATION_DELIVERED
        zuint48           u48CurrentTier3SummationDelivered;
    #endif

    #ifdef CLD_SM_ATTR_CURRENT_TIER_3_SUMMATION_RECEIVED
        zuint48           u48CurrentTier3SummationReceived;
    #endif

    #ifdef CLD_SM_ATTR_CURRENT_TIER_4_SUMMATION_DELIVERED
        zuint48           u48CurrentTier4SummationDelivered;
    #endif

    #ifdef CLD_SM_ATTR_CURRENT_TIER_4_SUMMATION_RECEIVED
        zuint48           u48CurrentTier4SummationReceived;
    #endif

    #ifdef CLD_SM_ATTR_CURRENT_TIER_5_SUMMATION_DELIVERED
        zuint48           u48CurrentTier5SummationDelivered;
```

```
#endif

#ifdef CLD_SM_ATTR_CURRENT_TIER_5_SUMMATION_RECEIVED
    zuint48            u48CurrentTier5SummationReceived;
#endif

#ifdef CLD_SM_ATTR_CURRENT_TIER_6_SUMMATION_DELIVERED
    zuint48            u48CurrentTier6SummationDelivered;
#endif

#ifdef CLD_SM_ATTR_CURRENT_TIER_6_SUMMATION_RECEIVED
    zuint48            u48CurrentTier6SummationReceived;
#endif

#ifdef CLD_SM_ATTR_CURRENT_TIER_7_SUMMATION_DELIVERED
    zuint48            u48CurrentTier7SummationDelivered;
#endif

#ifdef CLD_SM_ATTR_CURRENT_TIER_7_SUMMATION_RECEIVED
    zuint48            u48CurrentTier7SummationReceived;
#endif

#ifdef CLD_SM_ATTR_CURRENT_TIER_8_SUMMATION_DELIVERED
    zuint48            u48CurrentTier8SummationDelivered;
#endif

#ifdef CLD_SM_ATTR_CURRENT_TIER_8_SUMMATION_RECEIVED
    zuint48            u48CurrentTier8SummationReceived;
#endif

#ifdef CLD_SM_ATTR_CURRENT_TIER_9_SUMMATION_DELIVERED
    zuint48            u48CurrentTier9SummationDelivered;
#endif

#ifdef CLD_SM_ATTR_CURRENT_TIER_9_SUMMATION_RECEIVED
    zuint48            u48CurrentTier9SummationReceived;
#endif

#ifdef CLD_SM_ATTR_CURRENT_TIER_10_SUMMATION_DELIVERED
    zuint48            u48CurrentTier10SummationDelivered;
#endif

#ifdef CLD_SM_ATTR_CURRENT_TIER_10_SUMMATION_RECEIVED
    zuint48            u48CurrentTier10SummationReceived;
#endif

#ifdef CLD_SM_ATTR_CURRENT_TIER_11_SUMMATION_DELIVERED
    zuint48            u48CurrentTier11SummationDelivered;
#endif

#ifdef CLD_SM_ATTR_CURRENT_TIER_11_SUMMATION_RECEIVED
    zuint48            u48CurrentTier11SummationReceived;
#endif
```

```
#ifdef CLD_SM_ATTR_CURRENT_TIER_12_SUMMATION_DELIVERED
    zuint48            u48CurrentTier12SummationDelivered;
#endif

#ifdef CLD_SM_ATTR_CURRENT_TIER_12_SUMMATION_RECEIVED
    zuint48            u48CurrentTier12SummationReceived;
#endif

#ifdef CLD_SM_ATTR_CURRENT_TIER_13_SUMMATION_DELIVERED
    zuint48            u48CurrentTier13SummationDelivered;
#endif

#ifdef CLD_SM_ATTR_CURRENT_TIER_13_SUMMATION_RECEIVED
    zuint48            u48CurrentTier13SummationReceived;
#endif

#ifdef CLD_SM_ATTR_CURRENT_TIER_14_SUMMATION_DELIVERED
    zuint48            u48CurrentTier14SummationDelivered;
#endif

#ifdef CLD_SM_ATTR_CURRENT_TIER_14_SUMMATION_RECEIVED
    zuint48            u48CurrentTier14SummationReceived;
#endif

#ifdef CLD_SM_ATTR_CURRENT_TIER_15_SUMMATION_DELIVERED
    zuint48            u48CurrentTier15SummationDelivered;
#endif

#ifdef CLD_SM_ATTR_CURRENT_TIER_15_SUMMATION_RECEIVED
    zuint48            u48CurrentTier15SummationReceived;
#endif

    /* Meter status attribute set attribute ID's (D.3.2.2.3) */
    zbmap8             u8MeterStatus;                        /* Mandatory */

#ifdef CLD_SM_ATTR_REMAINING_BATTERY_LIFE
    uint8              u8RemainingBatteryLife;
#endif

#ifdef CLD_SM_ATTR_HOURS_IN_OPERATION
    zuint24            u24HoursInOperation;
#endif

#ifdef CLD_SM_ATTR_HOURS_IN_FAULT
    zuint24            u24HoursInFault;
#endif

    /* Formatting attribute set attribute ID's (D.3.2.2.4) */
    zenum8             eUnitOfMeasure;                        /* Mandatory */

#ifdef CLD_SM_ATTR_MULTIPLIER
    zuint24            u24Multiplier;
#endif
```

```
#ifdef CLD_SM_ATTR_DIVISOR
    zuint24             u24Divisor;
#endif

    zbmap8              u8SummationFormatting;                /* Mandatory */

#ifdef CLD_SM_ATTR_DEMAND_FORMATING
    zbmap8              u8DemandFormatting;
#endif

#ifdef CLD_SM_ATTR_HISTORICAL_CONSUMPTION_FORMATTING
    zbmap8              u8HistoricalConsumptionFormatting;
#endif

    zbmap8              eMeteringDeviceType;                /* Mandatory */

#ifdef CLD_SM_ATTR_SITE_ID
    tsZCL_OctetString  sSiteId;
    uint8              au8SiteId[SE_SM_SITE_ID_MAX_STRING_LENGTH];
#endif

#ifdef CLD_SM_ATTR_METER_SERIAL_NUMBER
    tsZCL_OctetString  sMeterSerialNumber;
    uint8              au8MeterSerialNumber[SE_SM_METER_SERIAL_NUMBER_MAX_STRING_LENGTH];
#endif

#ifdef CLD_SM_ATTR_ENERGY_CARRIER_UNIT_OF_MEASURE
    zenum8              e8EnergyCarrierUnitOfMeasure;
#endif

#ifdef CLD_SM_ATTR_ENERGY_CARRIER_SUMMATION_FORMATTING
    zbmap8              u8EnergyCarrierSummationFormatting;
#endif

#ifdef CLD_SM_ATTR_ENERGY_CARRIER_DEMAND_FORMATTING
    zbmap8              u8EnergyCarrierDemandFormatting;
#endif

#ifdef CLD_SM_ATTR_TEMPERATURE_UNIT_OF_MEASURE
    zenum8              e8TemperatureUnitOfMeasure;
#endif

#ifdef CLD_SM_ATTR_TEMPERATURE_FORMATTING
    zbmap8              u8TemperatureFormatting;
#endif

    /* ESP Historical Consumption set attribute ID's (D.3.2.2.5) */
#ifdef CLD_SM_ATTR_INSTANTANEOUS_DEMAND
    zint24              i24InstantaneousDemand;
#endif

#ifdef CLD_SM_ATTR_CURRENT_DAY_CONSUMPTION_DELIVERED
    zuint24             u24CurrentDayConsumptionDelivered;
#endif
```

```
#ifdef CLD_SM_ATTR_CURRENT_DAY_CONSUMPTION_RECEIVED
    zuint24            u24CurrentDayConsumptionReceived;
#endif

#ifdef CLD_SM_ATTR_PREVIOUS_DAY_CONSUMPTION_DELIVERED
    zuint24            u24PreviousDayConsumptionDelivered;
#endif

#ifdef CLD_SM_ATTR_PREVIOUS_DAY_CONSUMPTION_RECEIVED
    zuint24            u24PreviousDayConsumptionReceived;
#endif

#ifdef CLD_SM_ATTR_CURRENT_PARTIAL_PROFILE_INTERVAL_START_TIME_DELIVERED
    zutctime           utctCurrentPartialProfileIntervalStartTimeDelivered;
#endif

#ifdef CLD_SM_ATTR_CURRENT_PARTIAL_PROFILE_INTERVAL_START_TIME_RECEIVED
    zutctime           utctCurrentPartialProfileIntervalStartTimeReceived;
#endif

#ifdef CLD_SM_ATTR_CURRENT_PARTIAL_PROFILE_INTERVAL_VALUE_DELIVERED
    zuint24            u24CurrentPartialProfileIntervalValueDelivered;
#endif

#ifdef CLD_SM_ATTR_CURRENT_PARTIAL_PROFILE_INTERVAL_VALUE_RECEIVED
    zuint24            u24CurrentPartialProfileIntervalValueReceived;
#endif

#ifdef CLD_SM_ATTR_CURRENT_DAY_MAXIMUM_PRESSURE
    zuint48            u48CurrentDayMaxPressure;
#endif

#ifdef CLD_SM_ATTR_CURRENT_DAY_MINIMUM_PRESSURE
    zuint48            u48CurrentDayMinPressure;
#endif

#ifdef CLD_SM_ATTR_PREVIOUS_DAY_MAXIMUM_PRESSURE
    zuint48            u48PreviousDayMaxPressure;
#endif

#ifdef CLD_SM_ATTR_PREVIOUS_DAY_MINIMUM_PRESSURE
    zuint48            u48PreviousDayMinPressure;
#endif

#ifdef CLD_SM_ATTR_CURRENT_DAY_MAXIMUM_DEMAND
    zint24             i24CurrentDayMaxDemand;
#endif

#ifdef CLD_SM_ATTR_PREVIOUS_DAY_MAXIMUM_DEMAND
    zint24             i24PreviousDayMaxDemand;
#endif

#ifdef CLD_SM_ATTR_CURRENT_MONTH_MAXIMUM_DEMAND
```

```
    zint24             i24CurrentMonthMaxDemand;
#endif

#ifdef CLD_SM_ATTR_CURRENT_YEAR_MAXIMUM_DEMAND
    zint24             i24CurrentYearMaxDemand;
#endif

#ifdef CLD_SM_ATTR_CURRENT_DAY_MAXIMUM_ENERGY_CARRIER_DEMAND
    zint24             i24CurrentDayMaxEnergyCarrierDemand;
#endif

#ifdef CLD_SM_ATTR_PREVIOUS_DAY_MAXIMUM_ENERGY_CARRIER_DEMAND
    zint24             i24PreviousDayMaxEnergyCarrierDemand;
#endif

#ifdef CLD_SM_ATTR_CURRENT_MONTH_MAXIMUM_ENERGY_CARRIER_DEMAND
    zint24             i24CurrentMonthMaxEnergyCarrierDemand;
#endif

#ifdef CLD_SM_ATTR_CURRENT_MONTH_MINIMUM_ENERGY_CARRIER_DEMAND
    zint24             i24CurrentMonthMinEnergyCarrierDemand;
#endif

#ifdef CLD_SM_ATTR_CURRENT_YEAR_MAXIMUM_ENERGY_CARRIER_DEMAND
    zint24             i24CurrentYearMaxEnergyCarrierDemand;
#endif

#ifdef CLD_SM_ATTR_CURRENT_YEAR_MINIMUM_ENERGY_CARRIER_DEMAND
    zint24             i24CurrentYearMinEnergyCarrierDemand;
#endif

    /* Load Profile attribute set attribute ID's (D.3.2.2.6) */
#ifdef CLD_SM_ATTR_MAX_NUMBER_OF_PERIODS_DELIVERED
    zuint8             u8MaxNumberOfPeriodsDelivered;
#endif

    /* Supply Limit attribute set attribute ID's (D.3.2.2.7) */
#ifdef CLD_SM_ATTR_CURRENT_DEMAND_DELIVERED
    zuint24            u24CurrentDemandDelivered;
#endif

#ifdef CLD_SM_ATTR_DEMAND_LIMIT
    zuint24            u24DemandLimit;
#endif

#ifdef CLD_SM_ATTR_DEMAND_INTEGRATION_PERIOD
    zuint8             u8DemandIntegrationPeriod;
#endif

#ifdef CLD_SM_ATTR_NUMBER_OF_DEMAND_SUBINTERVALS
    zuint8             u8NumberOfDemandSubintervals;
#endif

    /* Block Information attribute set attribute ID's (D.3.2.2.8) */
    /* No Tier Block */
#if (CLD_SM_ATTR_NO_TIER_BLOCK_CURRENT_SUMMATION_DELIVERED_MAX_COUNT != 0)
    zuint48            au48CurrentNoTierBlockSummationDelivered
                      [CLD_SM_ATTR_NO_TIER_BLOCK_CURRENT_SUMMATION_DELIVERED_MAX_COUNT];
#endif

    /* Tier 1 Block Set */
#if ((CLD_SM_ATTR_NUM_OF_TIERS_CURRENT_SUMMATION_DELIVERED > 0)&&
     (CLD_SM_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_CURRENT_SUMMATION_DELIVERED != 0))
    zuint48            au48CurrentTier1BlockSummationDelivered
                      [CLD_SM_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_CURRENT_SUMMATION_DELIVERED];
#endif

    /* Tier 2 Block Set */
#if ((CLD_SM_ATTR_NUM_OF_TIERS_CURRENT_SUMMATION_DELIVERED > 1)&&
     (CLD_SM_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_CURRENT_SUMMATION_DELIVERED != 0))
    zuint48            au48CurrentTier2BlockSummationDelivered
                      [CLD_SM_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_CURRENT_SUMMATION_DELIVERED];
#endif

    /* Tier 3 Block Set */
#if ((CLD_SM_ATTR_NUM_OF_TIERS_CURRENT_SUMMATION_DELIVERED > 2)&&
     (CLD_SM_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_CURRENT_SUMMATION_DELIVERED != 0))
    zuint48            au48CurrentTier3BlockSummationDelivered
                      [CLD_SM_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_CURRENT_SUMMATION_DELIVERED];
#endif

    /* Tier 4 Block Set */
#if ((CLD_SM_ATTR_NUM_OF_TIERS_CURRENT_SUMMATION_DELIVERED > 3)&&
     (CLD_SM_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_CURRENT_SUMMATION_DELIVERED != 0))
    zuint48            au48CurrentTier4BlockSummationDelivered
                      [CLD_SM_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_CURRENT_SUMMATION_DELIVERED];
#endif

    /* Tier 5 Block Set */
#if ((CLD_SM_ATTR_NUM_OF_TIERS_CURRENT_SUMMATION_DELIVERED > 4)&&
     (CLD_SM_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_CURRENT_SUMMATION_DELIVERED != 0))
    zuint48            au48CurrentTier5BlockSummationDelivered
                      [CLD_SM_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_CURRENT_SUMMATION_DELIVERED];
#endif

    /* Tier 6 Block Set */
#if ((CLD_SM_ATTR_NUM_OF_TIERS_CURRENT_SUMMATION_DELIVERED > 5)&&
     (CLD_SM_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_CURRENT_SUMMATION_DELIVERED != 0))
    zuint48            au48CurrentTier6BlockSummationDelivered
                      [CLD_SM_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_CURRENT_SUMMATION_DELIVERED];
#endif

    /* Tier 7 Block Set */
#if ((CLD_SM_ATTR_NUM_OF_TIERS_CURRENT_SUMMATION_DELIVERED > 6)&&
     (CLD_SM_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_CURRENT_SUMMATION_DELIVERED != 0))
    zuint48            au48CurrentTier7BlockSummationDelivered
                      [CLD_SM_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_CURRENT_SUMMATION_DELIVERED];
#endif

    /* Tier 8 Block Set */
```

```
#if ((CLD_SM_ATTR_NUM_OF_TIERS_CURRENT_SUMMATION_DELIVERED > 7)&&
     (CLD_SM_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_CURRENT_SUMMATION_DELIVERED != 0))
    zuint48            au48CurrentTier8BlockSummationDelivered
                      [CLD_SM_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_CURRENT_SUMMATION_DELIVERED];
#endif

    /* Tier 9 Block Set */
#if ((CLD_SM_ATTR_NUM_OF_TIERS_CURRENT_SUMMATION_DELIVERED > 8)&&
     (CLD_SM_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_CURRENT_SUMMATION_DELIVERED != 0))
    zuint48            au48CurrentTier9BlockSummationDelivered
                      [CLD_SM_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_CURRENT_SUMMATION_DELIVERED];
#endif

    /* Tier 10 Block Set */
#if ((CLD_SM_ATTR_NUM_OF_TIERS_CURRENT_SUMMATION_DELIVERED > 9)&&
     (CLD_SM_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_CURRENT_SUMMATION_DELIVERED != 0))
    zuint48            au48CurrentTier10BlockSummationDelivered
                       [CLD_SM_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_CURRENT_SUMMATION_DELIVERED];
#endif

    /* Tier 11 Block Set */
#if ((CLD_SM_ATTR_NUM_OF_TIERS_CURRENT_SUMMATION_DELIVERED > 10)&&
     (CLD_SM_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_CURRENT_SUMMATION_DELIVERED != 0))
    zuint48            au48CurrentTier11BlockSummationDelivered
                      [CLD_SM_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_CURRENT_SUMMATION_DELIVERED];
#endif

    /* Tier 12 Block Set */
#if ((CLD_SM_ATTR_NUM_OF_TIERS_CURRENT_SUMMATION_DELIVERED > 11)&&
     (CLD_SM_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_CURRENT_SUMMATION_DELIVERED != 0))
    zuint48            au48CurrentTier12BlockSummationDelivered
                      [CLD_SM_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_CURRENT_SUMMATION_DELIVERED];
#endif

    /* Tier 13 Block Set */
#if ((CLD_SM_ATTR_NUM_OF_TIERS_CURRENT_SUMMATION_DELIVERED > 12)&&
     (CLD_SM_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_CURRENT_SUMMATION_DELIVERED != 0))
    zuint48            au48CurrentTier13BlockSummationDelivered
                      [CLD_SM_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_CURRENT_SUMMATION_DELIVERED];
#endif

    /* Tier 14 Block Set */
#if ((CLD_SM_ATTR_NUM_OF_TIERS_CURRENT_SUMMATION_DELIVERED > 13)&&
     (CLD_SM_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_CURRENT_SUMMATION_DELIVERED != 0))
    zuint48            au48CurrentTier14BlockSummationDelivered
                      [CLD_SM_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_CURRENT_SUMMATION_DELIVERED];
#endif

    /* Tier 15 Block Set */
#if ((CLD_SM_ATTR_NUM_OF_TIERS_CURRENT_SUMMATION_DELIVERED > 14)&&
     (CLD_SM_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_CURRENT_SUMMATION_DELIVERED != 0))
    zuint48            au48CurrentTier15BlockSummationDelivered
                      [CLD_SM_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_CURRENT_SUMMATION_DELIVERED];
#endif

    /* Alarm attribute set attribute ID's (D.3.2.2.9) */
#ifdef CLD_SM_ATTR_GENERIC_ALARM_MASK
```

```
    zbmap16            u16GenericAlarmMask;
#endif

#ifdef CLD_SM_ATTR_ELECTRICITY_ALARM_MASK
    zbmap32            u32ElectricityAlarmMask;
#endif

#ifdef CLD_SM_ATTR_PRESSURE_ALARM_MASK
    zbmap16            u16PressureAlarmMask;
#endif

#ifdef CLD_SM_ATTR_WATER_SPECIFIC_ALARM_MASK
    zbmap16            u16WaterSpecificAlarmMask;
#endif

#ifdef CLD_SM_ATTR_HEAT_AND_COOLING_ALARM_MASK
    zbmap16            u16HeatAndCoolingSpecificAlarmMask;
#endif

#ifdef CLD_SM_ATTR_GAS_ALARM_MASK
    zbmap16            u16GasAlarmMask;
#endif
} tsCLD_SimpleMetering;
```

where:

‘Reading Information’ Attribute Set

- `u48CurrentSummationDelivered` is the total amount of the measured resource (e.g. electrical energy) delivered to the premises so far, expressed in the units specified in `eUnitOfMeasure` and in the format specified in `u8SummationFormatting`
- The following are optional attributes and are fully described in the *ZigBee Smart Energy Profile Specification*:
  - `u48CurrentSummationReceived`
  - `u48CurrentMaxDemandDelivered`
  - `u48CurrentMaxDemandReceived`
  - `u48DFTSummation`
  - `u16DailyFreezeTime`
  - `i8PowerFactor`
  - `utctReadingSnapshotTime`
  - `utctCurrentMaxDemandDeliveredTime`
  - `utctCurrentMaxDemandReceivedTime`
- The following are optional attributes that relate to Fast Polling mode (*both attributes are not certifiable in SE 1.1.1 or earlier and are for future use*):
  - `u32DefaultUpdatePeriod` is the default poll-period, in seconds, that is used in updating metering data outside of fast polling episodes

  - `u8FastPollUpdatePeriod` is the minimum poll-period, in seconds, that can be used in updating metering data during fast polling episodes (should not be set to less than 2 seconds)

- The following are optional attributes and are fully described in the *ZigBee Smart Energy Profile Specification* (*these attributes are not certifiable in SE 1.1.1 or earlier and are for future use*):
  - `u48CurrentBlockPeriodConsumptionDelivered`
  - `u24DailyConsumptionTarget`
  - `e8CurrentBlock`
- The following are optional attributes that relate to the 'Get Profile' feature:
  - `eProfileIntervalPeriod` is the time-interval over which one set of consumption data will be collected
  - `u32IntervalReadReportingPeriod` is the time-interval, in minutes, after which a sleepy End Device should wake up to provide metering data
- The following are optional attributes are fully described in the *ZigBee Smart Energy Profile Specification* (*all these attributes except* `u8SupplyStatus` *are not certifiable in SE 1.1.1 or earlier and are for future use*):
  - `u16PresetReadingTime`
  - `u16VolumePerReport`
  - `u8FlowRestriction`
  - `u8SupplyStatus`
  - `u48CurrentInletEnergyCarrierSummation`
  - `u48CurrentOutletEnergyCarrierSummation`
  - `i16InletTemperature`
  - `i16OutletTemperature`
  - `i16ControlTemperature`
  - `i24CurrentInletEnergyCarrierDemand`
  - `i24CurrentOutletEnergyCarrierDemand`

‘Time-Of-Use (TOU) Information’ Attribute Set

- The following are optional attributes and are fully described in the *ZigBee Smart Energy Profile Specification* (*the attributes for tiers 7 to 15 are not certifiable in SE 1.1.1 or earlier and are for future use*):
  - `u48CurrentTier1SummationDelivered`
  - `u48CurrentTier1SummationReceived`
  - `u48CurrentTier2SummationDelivered`
  - `u48CurrentTier2SummationReceived`

    :
  - `u48CurrentTier15SummationDelivered`
  - `u48CurrentTier15SummationReceived`

‘Meter Status’ Attribute Set

- `u8MeterStatus` is an 8-bit bitmap representing the status of the meter. Enumerated masks are provided that correspond to the possible settings - see Section 8.10.2 *(this attribute is only certifiable for electricity meters in SE 1.1.1)*
- The following are optional attributes and are fully described in the *ZigBee Smart Energy Profile Specification* (*all the attributes are not certifiable in SE 1.1.1 or earlier and are for future use*):
  - `u8RemainingBatteryLife`
  - `u24HoursInOperation`
  - `u24HoursInFault`

‘Formatting’ Attribute Set

- `eUnitOfMeasure` indicates the unit of measure for the resource quantity contained above in `u48CurrentSummationDelivered` and below in `i24InstantaneousDemand`. Enumerations for the possible units are provided - see Section 8.10.3
- The following are optional attributes and are fully described in the *ZigBee Smart Energy Profile Specification*:
  - `u24Multiplier`
  - `u24Divisor`
- `u8SummationFormatting` indicates the formatting for the resource quantity contained above in `u48CurrentSummationDelivered`. Enumerations for the possible formats are provided - see Section 8.10.4
- The following are optional attributes and are fully described in the *ZigBee Smart Energy Profile Specification*:
  - `u8DemandFormatting`
  - `u8HistoricalConsumptionFormatting`
- `eMeteringDeviceType` indicates the type of Metering Device in terms of the resource type which it measures. Enumerations for the possible device types are provided - see Section 8.10.6

- The following pair of elements represents an optional attribute which identifies the location of a Metering Device (*this attribute is not certifiable in SE 1.1.1 or earlier and is for future use*):
  - `sSiteId` is a `tsZCL_OctetString` structure containing information on the site identifier. This element is paired with `au8SiteId` (below)
  - `au8SiteId` is an array containing the site identifier. This element is paired with `sSiteId` (above)

Note: This identifier is known in the UK as the M-PAN for electricity and MPRN for gas, and in South Africa as as the 'Stand Point'. The field is large enough to accommodate the number of characters typically used in the UK and Europe (16 digits).

- The following pair of elements represents an optional attribute, which indicates the serial number of a Metering Device (*this attribute is not certifiable in SE 1.1.1 or earlier and is for future use*):
  - `sMeterSerialNumber` is a `tsZCL_OctetString` structure containing information on the serial number of a Metering Device. This element is paired with `au8SiteId` (below)
  - `au8MeterSerialNumber` is an array containing the serial number of a Metering Device. This element is paired with `sMeterSerialNumber` (above)
- The following are optional attributes and are fully described in the *ZigBee Smart Energy Profile Specification* (*these attributes are not certifiable in SE 1.1.1 or earlier and are for future use*):
  - `e8EnergyCarrierUnitOfMeasure`
  - `u8EnergyCarrierSummationFormatting`
  - `u8EnergyCarrierDemandFormatting`
  - `e8TemperatureUnitOfMeasure`
  - `u8TemperatureFormatting`

‘Historical Consumption’ Attribute Set

- `i24InstantaneousDemand` is an optional attribute containing the current rate of consumption of the metered resource with respect to time. The unit of measure for the relevant resource is as specified in `eUnitOfMeasure`

  If this attribute is used, the metering application should update its value on a regular basis, between once every second and once every five seconds. The attribute value can be negative, meaning that the relevant resource is currently being supplied from the premises to the utility company - for example, the case of locally generated electricity from roof-mounted solar panels being supplied to the national grid.
- The following are optional attributes and are fully described in the *ZigBee Smart Energy Profile Specification*:

- `u24CurrentDayConsumptionDelivered`
- `u24CurrentDayConsumptionReceived`
- `u24PreviousDayConsumptionDelivered`
- `u24PreviousDayConsumptionReceived`
- `utctCurrentPartialProfileIntervalStartTimeDelivered`
- `utctCurrentPartialProfileIntervalStartTimeReceived`
- `u24CurrentPartialProfileIntervalValueDelivered`
- `u24CurrentPartialProfileIntervalValueReceived`

- The following are optional attributes and are fully described in the *ZigBee Smart Energy Profile Specification* (*these attributes are not certifiable in SE 1.1.1 or earlier and are for future use*):
  - `u48CurrentDayMaxPressure`
  - `u48CurrentDayMinPressure`
  - `u48PreviousDayMaxPressure`
  - `u48PreviousDayMinPressure`
  - `i24CurrentDayMaxDemand`
  - `i24PreviousDayMaxDemand`
  - `i24CurrentMonthMaxDemand`
  - `i24CurrentYearMaxDemand`
  - `i24CurrentDayMaxEnergyCarrierDemand`
  - `i24PreviousDayMaxEnergyCarrierDemand`
  - `i24CurrentMonthMaxEnergyCarrierDemand`
  - `i24CurrentMonthMinEnergyCarrierDemand`
  - `i24CurrentYearMaxEnergyCarrierDemand`
  - `i24CurrentYearMinEnergyCarrierDemand`

‘Load Profile Configuration’ Attribute Set

- `u8MaxNumberOfPeriodsDelivered` is an optional attribute from the Simple Metering ‘Load Profile Configuration’ attribute set and is fully described in the *ZigBee Smart Energy Profile Specification*.

‘Supply Limit’ Attribute Set

- The following are optional attributes and are fully described in the *ZigBee Smart Energy Profile Specification*:
  - `u24CurrentDemandDelivered`
  - `u24DemandLimit`
  - `u8DemandIntegrationPeriod`
  - `u8NumberOfDemandSubintervals`

‘Block Information’ Attribute Set

- The following are optional attributes and are fully described in the *ZigBee Smart Energy Profile Specification* (*these attributes are not certifiable in SE 1.1.1 or earlier and are for future use*):
  - `au48CurrentNoTierBlockSummationDelivered[CLD_SM_ATTR_NO_TIER_BLOCK_CURRENT_SUMMATION_DELIVERED_MAX_COUNT]`
  - `au48CurrentTier1BlockSummationDelivered[CLD_SM_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_CURRENT_SUMMATION_DELIVERED]`
  - `au48CurrentTier2BlockSummationDelivered[CLD_SM_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_CURRENT_SUMMATION_DELIVERED]`
  - `au48CurrentTier3BlockSummationDelivered[CLD_SM_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_CURRENT_SUMMATION_DELIVERED]`
  - `au48CurrentTier4BlockSummationDelivered[CLD_SM_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_CURRENT_SUMMATION_DELIVERED]`
  - `au48CurrentTier5BlockSummationDelivered[CLD_SM_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_CURRENT_SUMMATION_DELIVERED]`
  - `au48CurrentTier6BlockSummationDelivered[CLD_SM_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_CURRENT_SUMMATION_DELIVERED]`
  - `au48CurrentTier7BlockSummationDelivered[CLD_SM_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_CURRENT_SUMMATION_DELIVERED]`
  - `au48CurrentTier8BlockSummationDelivered[CLD_SM_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_CURRENT_SUMMATION_DELIVERED]`
  - `au48CurrentTier9BlockSummationDelivered[CLD_SM_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_CURRENT_SUMMATION_DELIVERED]`
  - `au48CurrentTier10BlockSummationDelivered[CLD_SM_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_CURRENT_SUMMATION_DELIVERED]`
  - `au48CurrentTier11BlockSummationDelivered[CLD_SM_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_CURRENT_SUMMATION_DELIVERED]`
  - `au48CurrentTier12BlockSummationDelivered[CLD_SM_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_CURRENT_SUMMATION_DELIVERED]`
  - `au48CurrentTier13BlockSummationDelivered[CLD_SM_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_CURRENT_SUMMATION_DELIVERED]`
  - `au48CurrentTier14BlockSummationDelivered[CLD_SM_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_CURRENT_SUMMATION_DELIVERED]`
  - `au48CurrentTier15BlockSummationDelivered[CLD_SM_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_CURRENT_SUMMATION_DELIVERED]`

8.3 Attribute Settings

The Simple Metering cluster contains both mandatory and optional attributes (see Section 8.2). The cluster structure is shown below with only the mandatory attributes (which are enabled by default):

```
typedef struct PACK
{
  zuint48                     u48CurrentSummationDelivered;
  zbmap8                      u8MeterStatus;
  teSE_UnitOfMeasure          eUnitOfMeasure;
  zbmap8                      u8SummationFormatting;
  teSE_MeteringDeviceType     eMeteringDeviceType;
} tsSE_SimpleMetering;
```

The mandatory attribute settings are outlined below.

eMeteringDeviceType

The element `eMeteringDeviceType` of the structure `tsSE_SimpleMetering` indicates the type of Metering Device in terms of the resource type which it measures: electricity, gas, water, heat, cooling or pressure. This attribute belongs to the cluster's Formatting attribute set.

Enumerated values are provided for the full range of possible metering devices - for example, E_CLD_SM_MDT_GAS for a gas meter. Enumerated values are also provided for devices that mirror a Metering Device - for example, E_CLD_SM_MDT_GAS_MIRRORED for the mirroring device of a gas meter. All of these enumerations are defined in the structure `teCLD_SM_MeteringDeviceType`, detailed in Section 8.10.6.

u8MeterStatus

The element `u8MeterStatus` of the structure `tsSE_SimpleMetering` indicates the current status of the device by means of an 8-bit value. This attribute has its own attribute set, Meter Status.

The status value is a bitmap with the bit representations indicated in the table below:

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Reserved	Service Disconnect Open *	Leak Detect	Power Quality	Power Failure	Tamper Detect	Battery Low	Check Meter

* Set to '1' when service to this site has been disconnected

A bit is set (to '1') to indicate the corresponding error or warning.

A number of macros are defined in the SE API to reflect the above bit settings - for example, E_CLD_SM_METER_STATUS_POWER_FAILURE_BIT contains the state of the Power Failure bit (Bit 3). There are also macros for masking off the appropriate bit - these macros are detailed in Section 8.10.2.

eUnitOfMeasure

The element `eUnitOfMeasure` of the structure `tsSE_SimpleMetering` indicates the unit of measure in which the relevant resource is metered, e.g. kiloWatt-hour for electricity. This attribute belongs to the cluster's Formatting attribute set.

Enumerated values are provided for the possible units of measure - for example, E_CLD_SM_UOM_CUBIC_METER for cubic metre (of gas or water). This example will also configure measurements to be expressed in binary/hex. However, enumerated values are also provided to configure measurements to be expressed in binary coded decimal - for example, E_CLD_SM_UOM_CUBIC_METER_BCD configures measurements in cubic metres and expressed in binary coded decimal. All of these enumerations are defined in the structure `teCLD_SM_UnitOfMeasure`, detailed in Section 8.10.3.

u8SummationFormatting

The element `u8SummationFormatting` of the structure `tsSE_SimpleMetering` is an 8-bit value indicating the position of the decimal point in the metered value (see u48CurrentSummationDelivered). This attribute belongs to the cluster's Formatting attribute set.

This value contains bit fields, as follows:

- **Bits 2-0:** 3-bit value indicating number of digits to right of point
- **Bits 6-3:** 3-bit value indicating number of digits to left of point
- **Bit 7:** Setting this bit (to '1') suppresses leading zeros

A number of macros are defined in the SE API to accommodate the above format information - these macros are detailed in Section 8.10.4.

u48CurrentSummationDelivered

The element `u48CurrentSummationDelivered` of the structure `tsSE_SimpleMetering` is a 48-bit value representing the total quantity consumed, so far, of the metered resource (e.g. electrical energy). This attribute belongs to the Reading Information attribute set.

The attribute value is interpreted with the aid of the elements `eUnitOfMeasure` and `u8SummationFormatting`, which indicate the unit of measure and the position of the decimal point respectively.

8.4 Remotely Reading Simple Metering Attributes

The SE API provides dedicated functions for remotely reading the Simple Metering attributes:

1. The application must first call **eSE_ReadMeterAttributes()** to submit a ‘read attributes’ request to the relevant remote endpoint. The resulting read process is as described for **eZCL_SendReadAttributesRequest()** in Section 4.5.
2. On receiving the ‘read attributes’ response, the event E_ZCL_CBET_READ_ATTRIBUTES_RESPONSE is generated, which causes the callback function for the local endpoint to be invoked. This callback function should include a call to **eSE_HandleReadAttributesResponse()** which checks whether all the Simple Metering attributes are included in the response. If the response is not complete, the function will re-send ‘read attributes’ requests until all attribute values have been obtained.

Note that read access to cluster attributes must be explicitly enabled at compile-time as described in Section 3.5.1.

8.5 Mirroring Metering Data

'Mirroring' is a facility that stores and provides access to metering data which originates from Metering Devices that sleep. A Metering Device cannot be accessed during periods of sleep and therefore its data cannot normally be read at these times. Mirroring involves holding the data from sleepy Metering Devices centrally on a server, allowing access to the data at all times.

Normally, the ESP (Co-ordinator) acts as the mirroring server. One or more sleepy Metering Devices (End Devices) can mirror their data on this server. A Metering Device must send its latest data to the mirroring server immediately before entering sleep mode. This is illustrated in Figure 14 below.

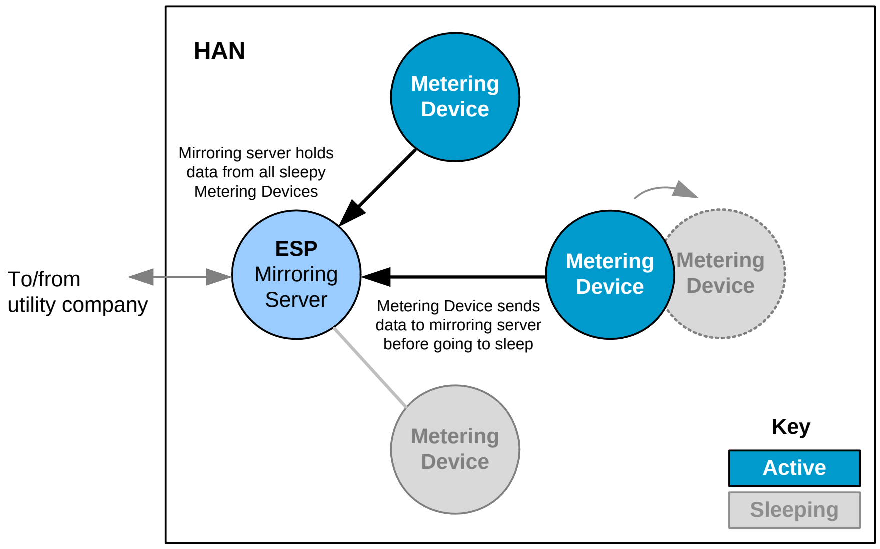
HAN
Metering Device
Mirroring server holds data from all sleepy Metering Devices
ESP Mirroring Server
To/from utility company
Metering Device
Metering Device
Metering Device sends data to mirroring server before going to sleep
Metering Device
Key
Active
Sleeping

Figure 14: Mirroring of Metering Data

Every mirror (one for each Metering Device) on the mirroring server has its own endpoint. The maximum number of mirror endpoints is defined at compile-time (see Section 8.12). Note that these endpoints are in addition to the main endpoint for the ESP (registered using **eSE_RegisterEspMeterEndPoint()** or **eSE_RegisterEspEndPoint()**).

Mirroring versions of the Simple Metering cluster server and/or client are implemented on the mirror endpoints. This is illustrated in Figure 15 below where the ESP, as the mirroring server, incorporates both the Simple Metering cluster server and client, the Metering device incorporates a cluster server and the IPD incorporates a cluster client.

The ESP device structure `tsSE_EspMeterDevice` (see Section 13.2.1) contains a section on mirroring support which includes an array of `tsSE_Mirror` structures (see Section 8.11.2). This array contains one element/structure per mirror endpoint, with the first mirror endpoint occupying array element 0 and the array size corresponding to the maximum number of mirror endpoints allowed on the mirroring server. The information stored in an array element includes the IEEE address of the Metering Device to which the mirror endpoint has been allocated.

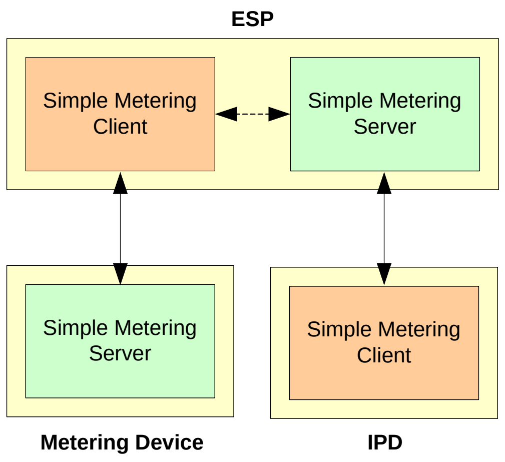

Figure 15: Simple Metering Cluster in Mirroring

8.5.1 Configuring Mirroring on ESP

The ESP normally acts as the mirroring server, containing a unique mirror endpoint for each (mirrored) Metering Device. Configuration of the mirroring server is carried out both within the application that runs on the device and as compile-time options - refer to Section 8.12 for the relevant compile-time options.

On the ESP, mirroring can be enabled in the application code when the device is registered using the function **eSE_RegisterEspMeterEndPoint()** or **eSE_RegisterEspEndPoint()**. These functions require specification of the first endpoint that is to be used for mirroring. Starting at this endpoint, consecutive endpoints to be used for mirrors will be reserved, up to the maximum number of mirrors defined by the compile-time option `CLD_SM_NUMBER_OF_MIRRORS`. For example, if 5 is specified as the first mirror endpoint and up to 4 mirrors can be used then endpoints 5, 6, 7 and 8 will be reserved for mirrors. Note that mirroring is disabled by setting the start endpoint to 0.

Note: The endpoints reserved for mirroring must also be included in the configuration diagram in the ZPS Configuration Editor. However, they must not be enabled since they will be enabled when mirrors are created on them.

The `tsSE_Mirror` structures in the ESP device structure `tsSE_EspMeterDevice` contain the IEEE addresses of the Metering Devices being mirrored on the ESP (these IEEE addresses are automatically initialised to zero). The ESP application must save an array of these IEEE addresses to non-volatile memory for persistent data storage, using the JenOS PDM module - this will allow the mirrored Metering Devices to be identified by the mirroring server following a reset of the ESP.

The ESP must allocate mirror endpoints to Metering Devices in response to requests from the Metering Devices (refer to Section 8.5.2 for details of requesting a mirror), as described below:

1. On receiving a mirror request on the ESP, the ZCL automatically allocates the next available mirror endpoint to the Metering Device (the IEEE address of the Metering Device is automatically written to the `tsSE_Mirror` structure which corresponds to the allocated mirror endpoint).
2. The event E_CLD_SM_CLIENT_RECEIVED_COMMAND containing the command E_CLD_SM_REQUEST_MIRROR is then generated on the ESP, causing the callback function on the ESP to be invoked.
3. The callback function must check whether all mirror endpoints have now been exhausted, in order to update the relevant status on the ESP. To do this, the function **eSM_GetFreeMirrorEndPoint()** must be called to obtain the number of the next free mirror endpoint. If the value 0xFFFF is returned, this means that no more mirror endpoints are available (for subsequent requests) and the attribute `u8PhysicalEnvironment` of the Basic cluster must be set to zero (to indicate to other Metering Devices that no more mirrors are available on the ESP). This step is illustrated in the code fragment below.

```
eSM_GetFreeMirrorEndPoint (&u16FoundEP);
if (u16FoundEP == 0xFFFF)
{
   psSE_EspMeterDevice->sBasicCluster.u8PhysicalEnvironment = 0x00;
}
else
{
   psSE_EspMeterDevice->sBasicCluster.u8PhysicalEnvironment = 0x01;
}
```

4. The callback function must copy the IEEE addresses from the `tsSE_Mirror` structures (which are automatically kept up-to-date) to the application's array of IEEE addresses for mirrored devices, and this array should be re-saved in non-volatile memory using the JenOS PDM module. This step is illustrated below in the code fragment under "Writing and Preserving Array of IEEE Addresses".
5. A response is automatically sent to the requesting Metering Device, where this response contains the number of the assigned endpoint.

The ESP is then ready to receive metering data from the remote Metering Device, as described in Section 8.5.3.

Writing and Preserving Array of IEEE Addresses

The ESP application must maintain an array of the IEEE addresses of the mirrored Metering Devices and keep a copy of this array in NVM. The array can be updated from the `tsSE_Mirror` structures for the mirror endpoints and saved to NVM as illustrated in the code fragment below:

```
case E_UPDATE_EVENT_REQUEST_MIRROR:
case E_UPDATE_EVENT_REMOVE_MIRROR:
{
  uint8 u8LoopCntr;
  for (u8LoopCntr =0; u8LoopCntr < CLD_SM_NUMBER_OF_MIRRORS; u8LoopCntr++)
      {
        sMirrorState.u64ExtAddr[u8LoopCntr] =
                     sMeter.sSE_Mirrors[u8LoopCntr].u64SourceAddress;
      }
      sMirrorState.bNetworkUp = TRUE;
      PDM_vSaveRecord(&g_sMirrorStateDescr);
}
break;
```

Note that `g_sMirrorStateDescr` maps to the `sMirrorState` structure.

Recreating Mirrors Following an ESP Reset

If the ESP is reset, the mirrors that have been created on the device will be lost. However, if the IEEE addresses (of the mirrored Metering Devices) associated with the mirror endpoints have been preserved in NVM, this data can be read by the ESP application following the reset and the mirrors recreated. Given the relevant endpoint number and IEEE address, a mirror can be recreated using the function **eSM_CreateMirror()**.

Note: A matching function **eSM_RemoveMirror()** also exists to allow the application to remove a mirror.

8.5.2 Configuring Mirroring on Metering Devices

Configuration of a Metering Device for mirroring is carried out both within the application that runs on the device and as a compile-time option - refer to Section 8.12 for the relevant compile-time options.

It is the responsibility of the Metering Device to request a mirror on the ESP, but first it must establish whether the ESP is accepting mirror requests. To do this, the application should use the function **eZCL_SendReadAttributesRequest()** to obtain the value of the `u8PhysicalEnvironment` attribute of the Basic cluster on the ESP - if this value is non-zero then the ESP is open to receiving mirror requests.

Provided that the ESP is accepting mirror requests, a Metering Device application can request a mirror using the function **eSM_ServerRequestMirrorCommand()**. This function sends a mirror request to the ESP with the aim of being allocated a mirror endpoint. The handling of this request on the ESP is described in Section 8.5.1.

The Metering Device application must then wait for a response from the ESP. This response is indicated by the event E_CLD_SM_SERVER_RECEIVED_COMMAND containing the command E_CLD_SM_REQUEST_MIRROR_RESPONSE, causing the callback function for the receiving endpoint to be invoked.

If the request has resulted in the successful allocation of a mirror endpoint on the ESP, the `tsSM_RequestMirrorResponseCommand` structure (see Section 8.11.6) in this event will contain the allocated endpoint number. In this case:

- The callback function should write the allocated endpoint number and mirroring server (ESP) IEEE address to non-volatile memory for persistent data storage using the JenOS PDM module.
- The Metering Device application can now send metering data for storage on the ESP whenever required, as described in Section 8.5.3.

Note: If the Metering Device subsequently requests another mirror on the same ESP, the same mirror endpoint number will be returned - a Metering Device cannot have more than one mirror on the same ESP.

If the request did not result in an allocated mirror endpoint on the ESP, the endpoint number returned in the above structure will be 0xFFFF and no action needs to be taken by the callback function.

8.5.3 Mirroring Data

Once a mirror for a Metering Device has been set up, as described in Section 8.5.1 and Section 8.5.2, the mirror can be populated and refreshed with data in two ways:

- The ESP application can submit a 'read attributes' request to the Metering Device (when it is not asleep), as described in Section 4.5.
- The Metering Device can send metering data as unsolicited attribute reports to the mirror at any time (for example, before entering sleep mode). This method is described further below.

The Metering Device application sends unsolicited attribute reports for the Simple Metering cluster to the mirror using the function **eZCL_ReportAllAttributes()**, described in the *ZCL User Guide (JN-UG-3077)*.

On receiving this data, the event E_ZCL_CBET_ATTRIBUTE_REPORT_MIRROR is generated on the ESP, causing the callback function on the ESP to be invoked. The callback function must then check that the data has come from a valid source (a Metering Device which has a mirror on the ESP) by calling the function **eSM_IsMirrorSourceAddressValid()**. According to the outcome of this check, the function updates the event status:

```
sZCL_CallBackEvent.uMessage.sReportAttributeMirror.eStatus
```

- If `eStatus` is set to E_ZCL_ATTR_REPORT_OK, the reported attribute values (metering data) are automatically stored on the relevant mirror endpoint and an E_ZCL_CBET_REPORT_INDIVIDUAL_ATTRIBUTE event is generated for each attribute reported.
- If `eStatus` is set to anything else, a ZCL default response is automatically sent back to the reporting device to indicate that mirroring is not authorised for this device (E_ZCL_CMDS_NOT_AUTHORIZED).

Maintaining the Mirrored eMeteringDeviceType Attribute

When a mirror is created on the ESP, the Simple Metering cluster attribute `eMeteringDeviceType` in the mirror will be set to the appropriate value for the Metering Device to be mirrored (e.g. E_CLD_SM_MDT_GAS). However, in order to distinguish the mirror cluster on the ESP from the original cluster on the Metering Device, the ESP application must replace this value in the mirror with the equivalent '_MIRRORED' value (e.g. E_CLD_SM_MDT_GAS_MIRRORED). In fact, this replacement must be performed every time the ESP receives a new set of attribute values from the Metering Device (by either of the two methods described above), since this attribute value in the mirror will be over-written each time and must subsequently be corrected.

8.5.4 Reading Mirrored Data

An SE device such as an IPD may need to obtain data from a mirror on the ESP, particularly when the mirrored Metering Device is sleeping. The data is requested by means of the standard 'read attributes' method, described in Section 4.5 - that is, by calling the ZCL function **eZCL_SendReadAttributesRequest()** on the requesting device.

If an attempt is made to read an attribute that currently has no value in the mirror, the resulting E_ZCL_CBET_READ_INDIVIDUAL_ATTRIBUTE_RESPONSE event will contain the attribute status E_ZCL_CMDS_UNSUPPORTED_ATTRIBUTE.

8.5.5 Removing a Mirror

The removal of a mirror on the ESP is initiated by the application on the corresponding Metering Device using the function **eSM_ServerRemoveMirrorCommand()**. This function sends a 'remove mirror' request to the relevant mirror endpoint on the ESP.

Note: A mirror can be removed from an endpoint on the ESP but the endpoint will remain reserved for mirroring - it may later be re-assigned to another mirror.

On receiving this request, the ESP processes the request as follows:

1. The ZCL first verifies the source address of the request to ensure that it has come from the Metering Device which corresponds to the mirror to be removed. If the source address is not valid then a ZCL default response is automatically sent to the requesting Metering Device to indicate that the request was not authorised (E_ZCL_CMDS_NOT_AUTHORIZED) - otherwise, the ESP continues to process the request as described in the steps below.
2. The ZCL then removes the mirror from the specified endpoint, thus freeing the endpoint for future use by another mirror.
3. The event E_CLD_SM_CLIENT_RECEIVED_COMMAND containing the command E_CLD_SM_REMOVE_MIRROR is generated on the ESP, causing the callback function on the ESP to be invoked.
4. The callback function must set the `u8PhysicalEnvironment` attribute of the Basic cluster to 0x01 in order to indicate that the ESP has the capacity to accept mirror requests (since the removal of the mirror leaves at least one mirror endpoint free).
5. The callback function must copy the IEEE addresses from the `tsSE_Mirror` structures (which are automatically kept up-to-date) to the application's array of IEEE addresses for mirrored devices, and this array should be re-saved in non-volatile memory using the JenOS PDM module. This step is illustrated in the code fragment under "Writing and Preserving Array of IEEE Addresses" on page 184.
6. A response is automatically sent to the requesting Metering Device to confirm the mirror removal.

The response (reporting successful mirror removal) results in the generation of the event E_CLD_SM_SERVER_RECEIVED_COMMAND containing the command E_CLD_SM_MIRROR_REMOVED on the Metering Device.

Note: The function **eSM_RemoveMirror()** is also provided, which allows the ESP application to directly remove a mirror.

8.6 Consumption Data Archive ('Get Profile')

Devices that support the Simple Metering cluster can maintain and exchange historical consumption (profiling) data using the 'Get Profile' feature. A consumption data archive, which is distinct from the data of the Simple Metering cluster attributes, is maintained in a circular buffer on the cluster server. A cluster client can make a 'Get Profile' request to the server to obtain data from this archive. Normally, the cluster server is implemented on a Metering Device and the cluster client is implemented on an IPD. Typically, the IPD requests a consumption history from the Metering Device in order to display this information to the consumer.

The consumption data in the archive corresponds to a series of consecutive time intervals with their corresponding consumption values. Thus, the archive consists of the last few consumption measurements - it is the responsibility of the application running on the server device to update the archive (see Section 8.6.1).

If the 'Get Profile' feature is required, it must be enabled in the compile-time options as described in Section 8.12. These options include the maximum number of consumption intervals that can be archived on the server (and therefore requested).

8.6.1 Updating Consumption Data on Server

The consumption archive is held on the Smart Metering cluster server in a circular buffer operating on a FIFO basis. This buffer provides storage space for a sequence of entries containing consumption data for consecutive time intervals, where each buffer entry is a structure of the type `tsSEGetProfile` consisting of:

- End-time of consumption interval (as UTC time)
- Units delivered to the customer
- Units received from the customer (when customer sells units to utility company)

The maximum number of entries that can be stored in the buffer is determined at compile-time (see Section 8.12). When a new entry is added to a full buffer, this entry replaces the oldest entry currently in the buffer.

The application must keep the buffer up-to-date by adding a new entry using the function **eSM_ServerUpdateConsumption()**. Before this function is called, the relevant consumption data must be updated in one or both of the following Simple Metering cluster attributes:

- `u24CurrentPartialProfileIntervalValueDelivered`

  Contains the number of units delivered to the customer over the last interval
- `u24CurrentPartialProfileIntervalValueReceived`

  Contains the number of units received from the customer over the last interval

An attribute only needs to be updated if the corresponding consumption has been implemented (for example, the utility company often only delivers units to the customer and does not receive any from the customer).

eSM_ServerUpdateConsumption() takes the current time as an input and then adds an entry containing the consumption data (in the above attributes) to the buffer, where the supplied current time becomes the end-time in the entry (thus, the duration of the consumption intervals is dictated by the frequency at which this function is called - see below).

Note: The current time can be obtained by the application using the function **u32ZCL_GetUTCTime()**, described in the *ZCL User Guide (JN-UG-3077)*.

eSM_ServerUpdateConsumption() must be called periodically by the application. The period must match the value to which the Simple Metering `eProfileIntervalPeriod` attribute has been set (see Section 8.2). Standard periods, ranging from 2.5 minutes to one day, are provided as a set of enumerations (see Section 8.10.10).

8.6.2 Sending and Handling a 'Get Profile' Request

The application on a device which supports the Simple Metering cluster as a client, such as an IPD, can send a 'Get Profile' request to the cluster server by calling the function **eSM_ClientGetProfileCommand()**. This function allows consumption data to be requested from the archive for one or more intervals.

The inputs for this function include:

- A value indicating whether the units delivered or units received (by the utility company) are being requested (see Section 8.6.1)
- An end-time (as a UTC time) - the most recent consumption data will be reported which has an end-time equal to or earlier than this end-time (a specified end-time of zero will result in the most recent consumption data)
- The number of consumption intervals to report (this number will be reported only if data for sufficient intervals is available) - the end-time rule, specified above, will be applied to all the reported intervals

On receiving the request, the event E_CLD_SM_SERVER_RECEIVED_COMMAND containing the command E_CLD_SM_GET_PROFILE is generated on the server, causing the callback function on the device to be invoked (for a Metering Device, this is the callback function registered through **eSE_RegisterEspMeterEndPoint()** or **eSE_RegisterMeterEndPoint()**). The callback function only needs to be concerned with this event if the archive data needs to be modified before the ZCL automatically sends the requested data in a 'Get Profile' response. The response indicates the number of consumption intervals reported and contains the consumption data for these intervals, as well as the end-time of the most recent interval reported.

On receiving the response, the event E_CLD_SM_CLIENT_RECEIVED_COMMAND containing the command E_CLD_SM_GET_PROFILE_RESPONSE is generated on the requesting client, causing the callback function on the device to be invoked (for an IPD, this is the callback function registered through **eSE_RegisterIPDEndPoint()**). The callback function should extract the requested data from the event using the function **u32SM_GetReceivedProfileData()** in order to process or store the data. This function should be called for each consumption interval reported in the event - the code fragment below illustrates repeated calls to the function until all the reported data has been obtained:

```
for (i =0 ;i <
sGetProfileResponseCommand.u8NumberOfPeriodsDelivered; i++)
{
//Read data from event
X(i) = u32SM_GetReceivedProfileData(tsSM_GetProfileResponseCommand
*psSMGetProfileResponseCommand)
}
```

Alternatively, the function can be called repeatedly until it returns 0xFFFFFFFF, which indicates that there is no more data to be extracted from the event.

8.7 Simple Metering Events

The Simple Metering cluster has its own events that are handled through the callback mechanism outlined in Section 4.7 (and fully detailed in the *ZCL User Guide (JN-UG-3077)*). If a device uses the Simple Metering cluster then Simple Metering event handling must be included in the callback function for the associated endpoint, where this callback function is registered through the relevant endpoint registration function (for example, through **eSE_RegisterMeterEndPoint()** for a standalone Metering Device). The relevant callback function will then be invoked when a Simple Metering event occurs.

For a Simple Metering event, the `eEventType` field of the `tsZCL_CallBackEvent` structure is set to E_ZCL_CBET_CLUSTER_CUSTOM. This event structure also contains an element `sClusterCustomMessage`, which is itself a structure containing a field `pvCustomData`. This field is a pointer to a `tsSM_CallBackMessage` structure which contains the Simple Metering parameters:

```
typedef struct
{
  teSM_CallBackEventType eEventType;
  uint8 u8CommandId;

    union
    {
       tsSM_GetProfileResponseCommand       sGetProfileResponseCommand;
       tsSM_RequestFastPollResponseCommand  sRequestFastPollResponseCommand;
       tsSM_GetProfileRequestCommand        sGetProfileCommand;
       tsSM_RequestMirrorResponseCommand    sRequestMirrorResponseCommand;
       tsSM_MirrorRemovedResponseCommand    sMirrorRemovedResponseCommand;
       tsSM_RequestFastPollCommand          sRequestFastPollCommand;
       tsSM_Error                           sError;
    }uMessage;
}tsSM_CallBackMessage;
```

Information on the elements of the above structure is provided below.

8.7.1 Event Types

The `eEventType` field of the `tsSM_CallBackMessage` structure specifies the type of Simple Metering event that has been generated. These event types are enumerated in the `teSM_CallBackEventType` structure (see Section 8.10.7) and are listed in the table below.

Event Type Enumeration	Description
E_CLD_SM_CLIENT_RECEIVED_COMMAND	Generated when a command has been received on a cluster client
E_CLD_SM_SERVER_RECEIVED_COMMAND	Generated when a command has been received on the cluster server
E_CLD_SM_FAST_POLLING_TIMER_EXPIRED	Generated on the cluster server at the end of a fast polling episode (*for future use*)

The possible command types for the above event types are listed in Section 8.7.2.

8.7.2 Command Types

For each event type listed in Section 8.7.1, one of a number of command types could have been received. The relevant command type is specified through the `u8CommandId` field of the `tsSM_CallBackMessage` structure. The possible command types for each event type are detailed below.

E_CLD_SM_CLIENT_RECEIVED_COMMAND

The E_CLD_SM_CLIENT_RECEIVED_COMMAND event is generated when a command has been received on a cluster client. The possible command types for this event type are listed in the table below, which gives the enumerations and the associated `uMessage` union elements in the `tsSM_CallBackMessage` structure:

u8CommandId Enumeration	uMessage Union Element
E_CLD_SM_GET_PROFILE_RESPONSE	sGetProfileResponseCommand
E_CLD_SM_REQUEST_MIRROR	sRequestMirrorAdd
E_CLD_SM_REMOVE_MIRROR	sRequestMirrorRemove
E_CLD_SM_REQUEST_FAST_POLL_MODE_ RESPONSE	sRequestFastPollResponseCommand (*for future use*)
E_CLD_SM_CLIENT_ERROR	sError

The above command enumerations are fully described in Section 8.10.8.

E_CLD_SM_SERVER_RECEIVED_COMMAND

The E_CLD_SM_SERVER_RECEIVED_COMMAND event is generated when a command has been received on the cluster server. The possible command types for this event type are listed in the table below, which gives the enumerations and the associated `uMessage` union elements in the `tsSM_CallBackMessage` structure:

u8CommandId Enumeration	uMessage Union Element
E_CLD_SM_GET_PROFILE	sGetProfileCommand
E_CLD_SM_REQUEST_MIRROR_RESPONSE	sRequestMirrorResponseCommand
E_CLD_SM_MIRROR_REMOVED	sMirrorRemovedResponseCommand
E_CLD_SM_REQUEST_FAST_POLL_MODE	sRequestFastPollCommand (*for future use*)
E_CLD_SM_SERVER_ERROR	sError

The above command enumerations are fully described in Section 8.10.9.

E_CLD_SM_FAST_POLLING_TIMER_EXPIRED

The E_CLD_SM_FAST_POLLING_TIMER_EXPIRED event is generated on the cluster server at the end of a fast polling episode. It has no associated data structure. *Fast polling is not certifiable in SE 1.1.1 or earlier and this event is reserved for future use*.

8.8 Functions

The following Simple Metering cluster functions are provided in the SE API:

Function	Page
eSE_SMCreate	194
eSE_ReadMeterAttributes	196
eSE_HandleReadMeterAttributesResponse	198
eSM_ServerRequestMirrorCommand	199
eSM_ServerRemoveMirrorCommand	201
eSM_CreateMirror	202
eSM_RemoveMirror	203
eSM_GetFreeMirrorEndPoint	204
eSM_IsMirrorSourceAddressValid	205
eSM_ServerUpdateConsumption	206
eSM_ClientGetProfileCommand	207
u32SM_GetReceivedProfileData	209

eSE_SMCreate

```
teZCL_Status eSE_SMCreate(
                    uint8 u8Endpoint,
                    bool_t  bIsServer,
                    uint8 *pu8AttributeControlBits,
                    tsZCL_ClusterInstance *psClusterInstance,
                    tsZCL_ClusterDefinition *psClusterDefinition,
                    tsSM_CustomStruct *psCustomDataStruct,
                    void *pvEndPointSharedStructPtr);
```

Description

This function creates an instance of the Simple Metering cluster on an endpoint. The cluster instance is created on the endpoint which is associated with the supplied `tsZCL_ClusterInstance` structure and can act as a server or a client, as specified.

The function should only be called when setting up a custom endpoint containing one or more selected clusters (rather than the whole set of clusters supported by a standard ZigBee device). This function will create a Simple Metering cluster instance on the endpoint, but instances of other clusters may also be created on the same endpoint by calling their corresponding creation functions. For more details of creating cluster instances on custom endpoints, refer to Appendix B.

Note: This function must not be called for an endpoint on which a standard ZigBee device (e.g. IPD) will be used. In this case, the device and its supported clusters must be registered on the endpoint using the relevant device registration function from those described in Chapter 12.

When used, this function must be the first Simple Metering cluster function called in the application, and must be called after the stack has been started and after the application profile has been initialised.

The function requires an array to be declared for internal use, which contains one element (of type **uint8**) for each attribute of the cluster. The array length should therefore equate be the total number of attributes supported by the Simple Metering cluster, which can be obtained by using the macro CLD_SM_MAX_NUMBER_OF_ATTRIBUTE.

The array declaration should be as follows:

```
uint8 au8AppSMClusterAttributeControlBits[CLD_SM_MAX_NUMBER_OF_ATTRIBUTE];
```

The function will initialise the array elements to zero.

Parameters

u8Endpoint	Number of local endpoint on which the cluster instance is to be created, in the range 1 to 240.
bIsServer	Type of cluster instance (server or client) to be created: TRUE - server FALSE - client
pu8AttributeControlBits	Pointer to an array of **uint8** values, with one element for each attribute in the cluster (see above).
psClusterInstance	Pointer to structure containing information about the cluster instance to be created (see the *ZCL User Guide (JN-UG-3077)*). This structure will be updated by the function by initialising individual structure fields.
psClusterDefinition	Pointer to structure indicating the type of cluster to be created (see the *ZCL User Guide (JN-UG-3077)*). In this case, this structure must contain the details of the Simple Metering cluster. This parameter can refer to a pre-filled structure called `sCLD_SimpleMetering` which is provided in the **SimpleMetering.h** file.
psCustomDataStructure	Pointer to structure which contains custom data for the Simple Metering cluster. This structure is used for internal data storage and also contains data relating to a received command/message. No knowledge of the fields of this structure is required.
pvEndPointSharedStructPtr	Pointer to the shared structure used for attribute storage. This parameter should be the address of the structure of type `tsCLD_SimpleMetering` which defines the attributes of Simple Metering cluster. The function will initialise the attributes with default values.

Returns

E_ZCL_SUCCESS

E_ZCL_FAIL

E_ZCL_ERR_PARAMETER_NULL

E_ZCL_ERR_INVALID_VALUE

eSE_ReadMeterAttributes

```
teZCL_Status eSE_ReadMeterAttributes(
                    uint8 u8SourceEndPointId,
                    uint8 u8DestinationEndPointId,
                    tsZCL_Address *psDestinationAddress,
                    uint8 *pu8TransactionSequenceNumber);
```

Description

This function can be used to send a ‘read attributes’ request to the Simple Metering cluster on a remote endpoint. The function requests all Simple Metering attributes to be read - alternatively, the function **eZCL_SendReadAttributesRequest()** can be used if only specific attributes are required. Note that read access to cluster attributes on the remote node (server) and local node (client) must be enabled at compile-time, as described in Section 3.5.1.

You must specify the endpoint on the local node from which the request is to be sent. This is also used to identify the instance of the local shared device structure which holds the relevant attributes. The obtained attribute values will be written to this shared structure by the function.

You must also specify the address of the destination node and the destination endpoint number. It is possible to use this function to send a request to bound endpoints or to a group of endpoints on remote nodes - in the latter case, a group address must be specified. Note that when sending requests to multiple endpoints through a single call to this function, multiple responses will subsequently be received from the remote endpoints.

You are also required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response will be set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

Following the first response to this function call, your application should call the function **eSE_HandleReadAttributesResponse()** to ensure that all the Simple Metering attributes are received from the remote endpoint.

Parameters

u8SourceEndPointId	Number of the local endpoint through which the request will be sent
u8DestinationEndPointId	Number of the remote endpoint to which the request will be sent. Note that this parameter is ignored when sending to address types E_ZCL_AM_BOUND and E_ZCL_AM_GROUP
psDestinationAddress	Pointer to a structure containing the address of the remote node to which the request will be sent
pu8TransactionSequenceNumber	Pointer to a location to store the Transaction Sequence Number (TSN) of the request

Returns

E_ZCL_SUCCESS
E_ZCL_ERR_CLUSTER_NOT_FOUND
E_ZCL_ERR_CLUSTER_ID_RANGE
E_ZCL_ERR_EP_UNKNOWN
E_ZCL_ERR_EP_RANGE
E_ZCL_ERR_ATTRIBUTE_WO
E_ZCL_ERR_ATTRIBUTES_ACCESS
E_ZCL_ERR_ATTRIBUTE_NOT_FOUND
E_ZCL_ERR_PARAMETER_NULL
E_ZCL_ERR_PARAMETER_RANGE

eSE_HandleReadMeterAttributesResponse

```
teSE_Status eSE_HandleReadAttributesResponse(
                        tsZCL_CallBackEvent *psEvent,
                        uint8 *puTransactionSequenceNumber);
```

Description

This function should be called after **eSE_ReadMeterAttributes()**. The function examines the response to a ‘read attributes’ request for the Simple Metering cluster and determines whether the response is complete - that is, whether it contains all the Simple Metering attributes (the response may be incomplete if the returned data is too large to fit into a single APDU). If the response is not complete, the function will re-send ‘read attributes’ requests until all attribute values have been obtained. Any further attribute values obtained will be written to the local shared device structure containing the attributes.

This function call should normally be included in the user-defined callback function that is invoked when the event E_ZCL_CBET_READ_ATTRIBUTES_RESPONSE is generated. This is the callback function which is specified when the (requesting) endpoint is registered using the appropriate endpoint registration function from Chapter 12. The callback function must pass the generated event into **eSE_HandleReadAttributesResponse()**.

You are also required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request/response.

Parameters

psEvent	Pointer to the generated event E_ZCL_CBET_READ_ATTRIBUTES_RESPONSE
pu8TransactionSequenceNumber	Pointer to a location to store the Transaction Sequence Number (TSN) of the request/ response

Returns

E_ZCL_SUCCESS

E_ZCL_ERR_CLUSTER_NOT_FOUND

E_ZCL_ERR_CLUSTER_ID_RANGE

E_ZCL_ERR_EP_UNKNOWN

E_ZCL_ERR_EP_RANGE

E_ZCL_ERR_ATTRIBUTE_WO

E_ZCL_ERR_ATTRIBUTES_ACCESS

E_ZCL_ERR_ATTRIBUTE_NOT_FOUND

E_ZCL_ERR_PARAMETER_NULL

E_ZCL_ERR_PARAMETER_RANGE

eSM_ServerRequestMirrorCommand

```
teZCL_Status eSM_ServerRequestMirrorCommand(
                    uint8 u8SourceEndpoint,
                    uint8 u8DestinationEndpoint,
                    tsZCL_Address *psDestinationAddress);
```

Description

This function can be used by a Metering Device to request a mirror on the ESP, for the central storage of its metering data. A mirror is useful for a Metering Device which sleeps, in order to allow access to its metering data while the device is sleeping.

The function sends an 'Add Mirror' request to the ESP. The address of the ESP device must be specified as well as the endpoint that will receive and process the request - this is the main endpoint on which the ESP is registered on the Co-ordinator. If successful, the request will result in the allocation of a mirror endpoint (on the ESP) to the Metering Device.

Note: Before using this function to send an 'Add Mirror' request, the Metering Device application should check whether the ESP is currently accepting these requests by calling the function **eZCL_SendReadAttributesRequest()** to obtain the value of the `u8PhysicalEnvironment` attribute of the Basic cluster on the ESP. This attribute value will be non-zero if 'Add Mirror' requests are being accepted.

eSM_ServerRequestMirrorCommand() is a non-blocking function and so returns immediately after the request has been sent. The application must then wait for a response, indicated by the event E_CLD_SM_SERVER_RECEIVED_COMMAND containing the command E_CLD_SM_REQUEST_MIRROR_RESPONSE. If a mirror was successfully created, the number of the allocated mirror endpoint on the ESP is included in the event.

Mirroring and mirror set-up are fully described in Section 8.5.

Parameters

u8SourceEndpoint	Number of local endpoint through which request will be sent
u8DestinationEndpoint	Number of ESP endpoint to which request will be sent (main endpoint of ESP)
psDestinationAddress	Pointer to a structure containing the address of the ESP device (to which the request will be sent)

Returns

E_ZCL_SUCCESS
E_ZCL_ERR_ZTRANSMIT_FAIL
E_ZCL_ERR_CLUSTER_NOT_FOUND
E_ZCL_ERR_CLUSTER_ID_RANGE
E_ZCL_ERR_EP_UNKNOWN
E_ZCL_ERR_EP_RANGE

eSM_ServerRemoveMirrorCommand

```
teZCL_Status eSM_ServerRemoveMirrorCommand(
                    uint8 u8SourceEndpoint,
                    uint8 u8DestinationEndpoint,
                    tsZCL_Address *psDestinationAddress);
```

Description

This function can be used on a Metering Device to request the removal of the corresponding mirror on the ESP. The function should only be used to remove a mirror that has been previously set up by the Metering Device application using the function **eSM_ServerRequestMirrorCommand()**.

The function sends a 'Remove Mirror' request to the ESP. The address of the ESP must be specified as well as the endpoint number of the mirror to be removed.

This is a non-blocking function and so returns immediately after the request has been sent. The application must then wait for a response.

- If the request was successful, a response will be received from the ESP resulting in the generation of the event E_CLD_SM_SERVER_RECEIVED_COMMAND containing the command E_CLD_SM_MIRROR_REMOVED
- If the request was unsuccessful, a ZCL default response will be received from the ESP to indicate that the request was not authorised (E_ZCL_CMDS_NOT_AUTHORIZED)

Mirror removal is fully described in Section 8.5.5.

Parameters

u8SourceEndpoint	Number of local endpoint through which request will be sent
u8DestinationEndpoint	Number of ESP endpoint which contains the mirror to be removed
psDestinationAddress	Pointer to a structure containing the address of the ESP device (to which the request will be sent)

Returns

E_ZCL_SUCCESS

E_ZCL_ERR_ZTRANSMIT_FAIL

E_ZCL_ERR_CLUSTER_NOT_FOUND

E_ZCL_ERR_CLUSTER_ID_RANGE

E_ZCL_ERR_EP_UNKNOWN

E_ZCL_ERR_EP_RANGE

eSM_CreateMirror

```
teSM_Status eSM_CreateMirror(
                        uint8 u8MirrorEndpoint,
                        uint64 u64RemoteIeeeAddress);
```

Description

This function can be used on the mirroring server (ESP) to create a mirror with the specified endpoint number for the Metering Device with the specified IEEE address. The endpoint number must be within the valid range for mirror endpoints on the ESP.

An error will be returned if there is no free mirror endpoint on which to create a mirror.

The function is normally used by an ESP application following a device reset, in order to recreate mirrors that were lost during the reset. This recovery assumes that the relevant IEEE addresses (for Metering Devices) associated with the mirror endpoints can be retrieved from non-volatile memory, where they were saved before the reset.

Parameters

u8MirrorEndpoint	Number of endpoint on which mirror will be created (must be within valid range for mirror endpoints)
u64RemoteIeeeAddress	IEEE address of Metering Device to be mirrored

Returns

E_ZCL_SUCCESS

E_ZCL_ERR_PARAMETER_RANGE

E_CLD_SM_STATUS_EP_NOT_AVAILABLE

eSM_RemoveMirror

```
teSM_Status eSM_RemoveMirror(
                          uint8 u8MirrorEndpoint,
                          uint64 u64RemoteIeeeAddress);
```

Description

This function can be used on the mirroring server (ESP) to remove the mirror with the specified endpoint number for the Metering Device with the specified IEEE address. The endpoint will then become free to be re-allocated for another mirror.

An error will be returned if the specified mirror endpoint cannot be found.

Parameters

u8MirrorEndpoint	Number of endpoint which hosts mirror to be removed (must be within valid range for mirror endpoints)
u64RemoteIeeeAddress	IEEE address of mirrored Metering Device

Returns

E_ZCL_SUCCESS

E_ZCL_ERR_PARAMETER_RANGE

E_CLD_SM_STATUS_EP_NOT_AVAILABLE

E_ZCL_FAIL

eSM_GetFreeMirrorEndPoint

```
teZCL_Status eSM_GetFreeMirrorEndPoint(
                                        uint16 *pu16FreeEP);
```

Description

This function can be used on the mirroring server (ESP) to obtain the number of the next available mirror endpoint. If there are no free mirror endpoints, the function sets the returned endpoint number to 0xFFFF.

The function is normally used in the ESP callback function to check the availability of mirror endpoints before updating the `u8PhysicalEnvironment` attribute of the Basic cluster (this attribute is set to zero if no more mirror endpoints are available).

Use of this function is described in Section 8.5.1.

Parameters

pu16FreeEP	Pointer to location to receive next free endpoint number

Returns

E_ZCL_SUCCESS

eSM_IsMirrorSourceAddressValid

```
eSM_IsMirrorSourceAddressValid(
  tsZCL_ReportAttributeMirror *psZCL_ReportAttributeMirror);
```

Description

This function can be used on the ESP to handle mirroring data reported from a Metering Device. If mirroring is enabled, the function should be included in the callback function on the ESP.

When the ESP receives mirroring data from a Metering Device, the event E_ZCL_CBET_ATTRIBUTE_REPORT_MIRROR is generated, causing the callback function to be invoked. The callback function should call this function to deal with the event.

The function first checks that the data comes from a Metering Device which has a mirror on the ESP (the source IEEE address of the data is used for this check) and then updates the event status accordingly:

`sZCL_CallBackEvent.uMessage.sReportAttributeMirror.eStatus`

If the source device is valid then this status is set to E_ZCL_ATTR_REPORT_OK and the metering data is automatically stored on the relevant mirror endpoint. Otherwise, a ZCL default response is returned to the Metering Device to indicate that mirroring is not authorised for this device (E_ZCL_CMDS_NOT_AUTHORIZED).

The mirroring of metering data is fully described in Section 8.5.3.

Parameters

psZCL_ReportAttributeMirror	Pointer to `sReportAttributeMirror` element of the event

Returns

E_ZCL_SUCCESS

eSM_ServerUpdateConsumption

```
teZCL_Status eSM_ServerUpdateConsumption(
                    uint8 u8SourceEndPointId,
                    uint32 u32UtcTime);
```

Description

This function can be used on a Simple Metering cluster server (with the 'Get Profile' feature enabled) to add a new entry to the circular buffer used to store historical consumption data. The buffer stores a sequence of entries containing consumption data for consecutive time intervals, identified by their end-times.

Before this function is called, the application must update one or both of the following Simple Metering cluster attributes with the relevant consumption(s) over the last time interval (since the last readings were made):

- `u24CurrentPartialProfileIntervalValueDelivered`
- `u24CurrentPartialProfileIntervalValueReceived`

An attribute only needs to be updated if the corresponding consumption has been implemented.

The function takes the current time (UTC time) as an input and adds a buffer entry containing the consumption measurements together with the supplied UTC time, which is saved as the end-time of the interval

The entry is stored as a `tsSEGetProfile` structure, described in Section 8.11.5.

The buffer can contain a limited number of entries, determined at compile-time (see Section 8.12), and operates on a FIFO basis so that a new entry added to a full buffer will over-write the oldest entry.

The function should be called periodically by the application. The period must match the value to which the Simple Metering `eProfileIntervalPeriod` attribute has been set (see Section 8.2). Standard periods, ranging from 2.5 minutes to one day, are provided as a set of enumerations (see Section 8.10.10).

Parameters

u8SourceEndPointId	Number of local endpoint on which the Simple Metering cluster server operates
u32UtcTime	Current time (as UTC time which can be obtained using **u32ZCL_GetUTCTime()**)

Returns

E_ZCL_SUCCESS

eSM_ClientGetProfileCommand

```
teZCL_Status eSM_ClientGetProfileCommand(
                        uint8 u8SourceEndpoint,
                        uint8 u8DestinationEndpoint,
                        tsZCL_Address *psDestinationAddress,
                        uint8 u8IntervalChannel,
                        uint32 u32EndTime,
                        uint8 u8NumberOfPeriods);
```

Description

This function can be used on a Simple Metering cluster client (with the 'Get Profile' feature enabled) to send a 'Get Profile' request to the Simple Metering cluster server in order to retrieve historical consumption data.

The server contains a circular buffer which stores a sequence of data entries containing consumption data for consecutive time intervals, identified by their end-times. This function can request a number of entries from the buffer, containing the consumption data over multiple intervals.

The function parameters include:

- The number of buffer entries (corresponding to consumption intervals) requested
- The most recent end-time for which a buffer entry will be reported - the most recent consumption data will be reported which has an end-time equal to or earlier than this end-time (a specified end-time of zero will result in the most recent consumption data)
- A value indicating whether the units delivered or units received are being requested

This is a non-blocking function and so returns immediately after the request has been sent. The application must then wait for a response, which is accessed using the function **u32SM_GetReceivedProfileData()**.

Parameters

u8SourceEndpoint	Number of local endpoint through which request will be sent
u8DestinationEndpoint	Number of endpoint to which request will be sent on the destination device
psDestinationAddress	Pointer to a structure containing the address of the destination device
u8IntervalChannel	Required consumption data - received or delivered: E_CLD_SM_CONSUMPTION_RECEIVED E_CLD_SM_CONSUMPTION_DELIVERED
u32EndTime	A UTC time representing the most recent interval end-time for which data will be reported (a zero value means report data for the most recent interval)
u8NumberOfPeriods	Number of consumption intervals to be reported

Returns

E_ZCL_SUCCESS
E_ZCL_ERR_ZTRANSMIT_FAIL
E_ZCL_ERR_CLUSTER_NOT_FOUND
E_ZCL_ERR_CLUSTER_ID_RANGE
E_ZCL_ERR_EP_UNKNOWN
E_ZCL_ERR_EP_RANGE

u32SM_GetReceivedProfileData

```
uint32 u32SM_GetReceivedProfileData(
            tsSM_GetProfileResponseCommand
                              *psSMGetProfileResponseCommand);
```

Description

This function can be used on a Simple Metering cluster client to obtain the consumption data received in a 'Get Profile' response from the Simple Metering cluster server (and previously requested by the client through a call to **eSM_ClientGetProfileCommand()**).

When a 'Get Profile' response from the server arrives, the event E_CLD_SM_CLIENT_RECEIVED_COMMAND containing the command E_CLD_SM_GET_PROFILE_RESPONSE is generated on the client, causing the callback function on the device to be invoked (for an IPD, this is the callback function registered through **eSE_RegisterIPDEndPoint()**). The callback function should deal with the response.

This function can be called within the callback function to extract consumption data from the event. It is necessary to provide the function with a pointer to the response within the event. The function will return the data corresponding to one consumption interval on each call. Therefore, if the response contains data for multiple intervals, the function must be called multiple times to extract all of this data - the number of intervals contained in the response is also included in the response:

```
sSMCallBackMessage.uMessage.sGetProfileResponseCommand.u8NumberOfPeriodsDelivered
```

When there is no more data to be extracted from the event, the function will return 0xFFFFFFFF.

Parameters

psSMGetProfileResponseCommand	Pointer to `sGetProfileResponseCommand` element of the event

Returns

32-bit value corresponding to consumption data for one interval

0xFFFFFFFF indicates that there is no more data to be read from the event

8.9 Return Codes

The Simple Metering cluster functions use the ZCL return codes defined in the *ZCL User Guide (JN-UG-3077)*.

8.10 Enumerations

8.10.1 ‘Attribute ID’ Enumerations

The following structure contains the enumerations used to identify the attributes of the Simple Metering cluster.

Note: Some of the following enumerations correspond to attributes that are not certifiable in SE 1.1.1 (07-5356-17) or earlier and are for future use (as indicated in the attribute descriptions in Section 8.2).

```
typedef enum PACK
{
    /* Reading information attribute set attribute ID's (D.3.2.2.1) */
    E_CLD_SM_ATTR_ID_CURRENT_SUMMATION_DELIVERED        =    0x0000,
    E_CLD_SM_ATTR_ID_CURRENT_SUMMATION_RECEIVED,
    E_CLD_SM_ATTR_ID_CURRENT_MAX_DEMAND_DELIVERED,
    E_CLD_SM_ATTR_ID_CURRENT_MAX_DEMAND_RECEIVED,
    E_CLD_SM_ATTR_ID_DFT_SUMMATION,
    E_CLD_SM_ATTR_ID_DAILY_FREEZE_TIME,
    E_CLD_SM_ATTR_ID_POWER_FACTOR,
    E_CLD_SM_ATTR_ID_READING_SNAPSHOT_TIME,
    E_CLD_SM_ATTR_ID_CURRENT_MAX_DEMAND_DELIVERED_TIME,
    E_CLD_SM_ATTR_ID_CURRENT_MAX_DEMAND_RECEIVED_TIME,
    E_CLD_SM_ATTR_ID_DEFAULT_UPDATE_PERIOD,
    E_CLD_SM_ATTR_ID_FAST_POLL_UPDATE_PERIOD,
    E_CLD_SM_ATTR_ID_CURRENT_BLOCK_PERIOD_CONSUMPTION_DELIVERED,
    E_CLD_SM_ATTR_ID_DAILY_CONSUMPTION_TARGET,
    E_CLD_SM_ATTR_ID_CURRENT_BLOCK,
    E_CLD_SM_ATTR_ID_PROFILE_INTERVAL_PERIOD,
    E_CLD_SM_ATTR_ID_INTERVAL_READ_REPORTING_PERIOD,
    E_CLD_SM_ATTR_ID_PRESET_READING_TIME,
    E_CLD_SM_ATTR_ID_VOLUME_PER_REPORT,
    E_CLD_SM_ATTR_ID_FLOW_RESTRICTION,
    E_CLD_SM_ATTR_ID_SUPPLY_STATUS,
    E_CLD_SM_ATTR_ID_CURRENT_INLET_ENERGY_CARRIER_SUMMATION,
    E_CLD_SM_ATTR_ID_CURRENT_OUTLET_ENERGY_CARRIER_SUMMATION,
    E_CLD_SM_ATTR_ID_INLET_TEMPERATURE,
    E_CLD_SM_ATTR_ID_OUTLET_TEMPERATURE,
    E_CLD_SM_ATTR_ID_CONTROL_TEMPERATURE,
    E_CLD_SM_ATTR_ID_CURRENT_INLET_ENERGY_CARRIER_DEMAND,
    E_CLD_SM_ATTR_ID_CURRENT_OUTLET_ENERGY_CARRIER_DEMAND,

    /* Time Of Use Information attribute attribute ID's set (D.3.2.2.2) */
    E_CLD_SM_ATTR_ID_CURRENT_TIER_1_SUMMATION_DELIVERED    =    0x0100,
    E_CLD_SM_ATTR_ID_CURRENT_TIER_1_SUMMATION_RECEIVED,
    E_CLD_SM_ATTR_ID_CURRENT_TIER_2_SUMMATION_DELIVERED,
    E_CLD_SM_ATTR_ID_CURRENT_TIER_2_SUMMATION_RECEIVED,
    :
    E_CLD_SM_ATTR_ID_CURRENT_TIER_15_SUMMATION_DELIVERED,
    E_CLD_SM_ATTR_ID_CURRENT_TIER_15_SUMMATION_RECEIVED,

    /* Meter status attribute set attribute ID's (D.3.2.2.3) */
    E_CLD_SM_ATTR_ID_STATUS                                 =    0x0200,
    E_CLD_SM_ATTR_ID_REMAINING_BATTERY_LIFE,
    E_CLD_SM_ATTR_ID_HOURS_IN_OPERATION,
    E_CLD_SM_ATTR_ID_HOURS_IN_FAULT,

    /* Formatting attribute set attribute ID's (D.3.2.2.4) */
    E_CLD_SM_ATTR_ID_UNIT_OF_MEASURE                        =    0x0300,
    E_CLD_SM_ATTR_ID_MULTIPLIER,
    E_CLD_SM_ATTR_ID_DIVISOR,
    E_CLD_SM_ATTR_ID_SUMMATION_FORMATING,
    E_CLD_SM_ATTR_ID_DEMAND_FORMATING,
    E_CLD_SM_ATTR_ID_HISTORICAL_CONSUMPTION_FORMATTING,
    E_CLD_SM_ATTR_ID_METERING_DEVICE_TYPE,
    E_CLD_SM_ATTR_ID_SITE_ID,
    E_CLD_SM_ATTR_ID_METER_SERIAL_NUMBER,
    E_CLD_SM_ATTR_ID_ENERGY_CARRIER_UNIT_OF_MEASURE,
    E_CLD_SM_ATTR_ID_ENERGY_CARRIER_SUMMATION_FORMATTING,
    E_CLD_SM_ATTR_ID_ENERGY_CARRIER_DEMAND_FORMATTING,
    E_CLD_SM_ATTR_ID_TEMPERATURE_UNIT_OF_MEASURE,
    E_CLD_SM_ATTR_ID_TEMPERATURE_FORMATTING,

    /* ESP Historical Consumption set attribute ID's (D.3.2.2.5) */
    E_CLD_SM_ATTR_ID_INSTANTANEOUS_DEMAND                 =    0x0400,
    E_CLD_SM_ATTR_ID_CURRENT_DAY_CONSUMPTION_DELIVERED,
    E_CLD_SM_ATTR_ID_CURRENT_DAY_CONSUMPTION_RECEIVED,
    E_CLD_SM_ATTR_ID_PREVIOUS_DAY_CONSUMPTION_DELIVERED,
    E_CLD_SM_ATTR_ID_PREVIOUS_DAY_CONSUMPTION_RECEIVED,
    E_CLD_SM_ATTR_ID_CURRENT_PARTIAL_PROFILE_INTERVAL_START_TIME_DELIVERED,
    E_CLD_SM_ATTR_ID_CURRENT_PARTIAL_PROFILE_INTERVAL_START_TIME_RECEIVED,
    E_CLD_SM_ATTR_ID_CURRENT_PARTIAL_PROFILE_INTERVAL_VALUE_DELIVERED,
    E_CLD_SM_ATTR_ID_CURRENT_PARTIAL_PROFILE_INTERVAL_VALUE_RECEIVED,
    E_CLD_SM_ATTR_ID_CURRENT_DAY_MAXIMUM_PRESSURE,
    E_CLD_SM_ATTR_ID_CURRENT_DAY_MINIMUM_PRESSURE,
    E_CLD_SM_ATTR_ID_PREVIOUS_DAY_MAXIMUM_PRESSURE,
    E_CLD_SM_ATTR_ID_PREVIOUS_DAY_MINIMUM_PRESSURE,
    E_CLD_SM_ATTR_ID_CURRENT_DAY_MAXIMUM_DEMAND,
    E_CLD_SM_ATTR_ID_PREVIOUS_DAY_MAXIMUM_DEMAND,
    E_CLD_SM_ATTR_ID_CURRENT_MONTH_MAXIMUM_DEMAND,
    E_CLD_SM_ATTR_ID_CURRENT_YEAR_MAXIMUM_DEMAND,
    E_CLD_SM_ATTR_ID_CURRENT_DAY_MAXIMUM_ENERGY_CARRIER_DEMAND,
    E_CLD_SM_ATTR_ID_PREVIOUS_DAY_MAXIMUM_ENERGY_CARRIER_DEMAND,
    E_CLD_SM_ATTR_ID_CURRENT_MONTH_MAXIMUM_ENERGY_CARRIER_DEMAND,
    E_CLD_SM_ATTR_ID_CURRENT_MONTH_MINIMUM_ENERGY_CARRIER_DEMAND,
    E_CLD_SM_ATTR_ID_CURRENT_YEAR_MAXIMUM_ENERGY_CARRIER_DEMAND,
    E_CLD_SM_ATTR_ID_CURRENT_YEAR_MINIMUM_ENERGY_CARRIER_DEMAND,

    /* Load Profile attribute set attribute ID's (D.3.2.2.6) */
    E_CLD_SM_ATTR_ID_MAX_NUMBER_OF_PERIODS_DELIVERED     =    0x0500,

    /* Supply Limit attribute set attribute ID's (D.3.2.2.7) */
    E_CLD_SM_ATTR_ID_CURRENT_DEMAND_DELIVERED            =    0x0600,
    E_CLD_SM_ATTR_ID_DEMAND_LIMIT,
    E_CLD_SM_ATTR_ID_DEMAND_INTEGRATION_PERIOD,
    E_CLD_SM_ATTR_ID_NUMBER_OF_DEMAND_SUBINTERVALS,

    /* Block Information Attribute set attribute ID's (D.3.2.2.8) */
    /* Block Information Attribute set: No Tier Block Set */
    E_CLD_SM_ATTR_ID_CURRENT_NOTIER_BLOCK1_SUMMATION_DELIVERED  =    0x0700,
    E_CLD_SM_ATTR_ID_CURRENT_NOTIER_BLOCK2_SUMMATION_DELIVERED,
    :
    E_CLD_SM_ATTR_ID_CURRENT_NOTIER_BLOCK16_SUMMATION_DELIVERED,

    /* Block Information Attribute set: Tier1 Block Set */
    E_CLD_SM_ATTR_ID_CURRENT_TIER1_BLOCK1_SUMMATION_DELIVERED,
    E_CLD_SM_ATTR_ID_CURRENT_TIER1_BLOCK2_SUMMATION_DELIVERED,
    :
    E_CLD_SM_ATTR_ID_CURRENT_TIER1_BLOCK16_SUMMATION_DELIVERED,

    /* Block Information Attribute set: Tier2 Block Set */
    E_CLD_SM_ATTR_ID_CURRENT_TIER2_BLOCK1_SUMMATION_DELIVERED,
    E_CLD_SM_ATTR_ID_CURRENT_TIER2_BLOCK2_SUMMATION_DELIVERED,
    :
    E_CLD_SM_ATTR_ID_CURRENT_TIER2_BLOCK16_SUMMATION_DELIVERED,

    /* Block Information Attribute set: Tier5 Block Set */
    E_CLD_SM_ATTR_ID_CURRENT_TIER3_BLOCK1_SUMMATION_DELIVERED,
    E_CLD_SM_ATTR_ID_CURRENT_TIER3_BLOCK2_SUMMATION_DELIVERED,
    :
    E_CLD_SM_ATTR_ID_CURRENT_TIER3_BLOCK16_SUMMATION_DELIVERED,

    /* Block Information Attribute set: Tier4 Block Set */
    E_CLD_SM_ATTR_ID_CURRENT_TIER4_BLOCK1_SUMMATION_DELIVERED,
    E_CLD_SM_ATTR_ID_CURRENT_TIER4_BLOCK2_SUMMATION_DELIVERED,
    :
```

```
E_CLD_SM_ATTR_ID_CURRENT_TIER4_BLOCK16_SUMMATION_DELIVERED,

/* Block Information Attribute set: Tier5 Block Set */
E_CLD_SM_ATTR_ID_CURRENT_TIER5_BLOCK1_SUMMATION_DELIVERED,
E_CLD_SM_ATTR_ID_CURRENT_TIER5_BLOCK2_SUMMATION_DELIVERED,
:
E_CLD_SM_ATTR_ID_CURRENT_TIER5_BLOCK16_SUMMATION_DELIVERED,

/* Block Information Attribute set: Tier6 Block Set */
E_CLD_SM_ATTR_ID_CURRENT_TIER6_BLOCK1_SUMMATION_DELIVERED,
E_CLD_SM_ATTR_ID_CURRENT_TIER6_BLOCK2_SUMMATION_DELIVERED,
:
E_CLD_SM_ATTR_ID_CURRENT_TIER6_BLOCK16_SUMMATION_DELIVERED,

/* Block Information Attribute set: Tier8 Block Set */
E_CLD_SM_ATTR_ID_CURRENT_TIER7_BLOCK1_SUMMATION_DELIVERED,
E_CLD_SM_ATTR_ID_CURRENT_TIER7_BLOCK2_SUMMATION_DELIVERED,
:
E_CLD_SM_ATTR_ID_CURRENT_TIER7_BLOCK16_SUMMATION_DELIVERED,

/* Block Information Attribute set: Tier8 Block Set */
E_CLD_SM_ATTR_ID_CURRENT_TIER8_BLOCK1_SUMMATION_DELIVERED,
E_CLD_SM_ATTR_ID_CURRENT_TIER8_BLOCK2_SUMMATION_DELIVERED,
:
E_CLD_SM_ATTR_ID_CURRENT_TIER8_BLOCK16_SUMMATION_DELIVERED,

/* Block Information Attribute set: Tier9 Block Set */
E_CLD_SM_ATTR_ID_CURRENT_TIER9_BLOCK1_SUMMATION_DELIVERED,
E_CLD_SM_ATTR_ID_CURRENT_TIER9_BLOCK2_SUMMATION_DELIVERED,
:
E_CLD_SM_ATTR_ID_CURRENT_TIER9_BLOCK16_SUMMATION_DELIVERED,

/* Block Information Attribute set: Tier10 Block Set */
E_CLD_SM_ATTR_ID_CURRENT_TIER10_BLOCK1_SUMMATION_DELIVERED,
E_CLD_SM_ATTR_ID_CURRENT_TIER10_BLOCK2_SUMMATION_DELIVERED,
:
E_CLD_SM_ATTR_ID_CURRENT_TIER10_BLOCK16_SUMMATION_DELIVERED,

/* Block Information Attribute set: Tier11 Block Set */
E_CLD_SM_ATTR_ID_CURRENT_TIER11_BLOCK1_SUMMATION_DELIVERED,
E_CLD_SM_ATTR_ID_CURRENT_TIER11_BLOCK2_SUMMATION_DELIVERED,
:
E_CLD_SM_ATTR_ID_CURRENT_TIER11_BLOCK16_SUMMATION_DELIVERED,

/* Block Information Attribute set: Tier12 Block Set */
E_CLD_SM_ATTR_ID_CURRENT_TIER12_BLOCK1_SUMMATION_DELIVERED,
E_CLD_SM_ATTR_ID_CURRENT_TIER12_BLOCK2_SUMMATION_DELIVERED,
:
```

```
            E_CLD_SM_ATTR_ID_CURRENT_TIER12_BLOCK16_SUMMATION_DELIVERED,

            /* Block Information Attribute set: Tier13 Block Set */
            E_CLD_SM_ATTR_ID_CURRENT_TIER13_BLOCK1_SUMMATION_DELIVERED,
            E_CLD_SM_ATTR_ID_CURRENT_TIER13_BLOCK2_SUMMATION_DELIVERED,
            :
            E_CLD_SM_ATTR_ID_CURRENT_TIER13_BLOCK16_SUMMATION_DELIVERED,

            /* Block Information Attribute set: Tier14 Block Set */
            E_CLD_SM_ATTR_ID_CURRENT_TIER14_BLOCK1_SUMMATION_DELIVERED,
            E_CLD_SM_ATTR_ID_CURRENT_TIER14_BLOCK2_SUMMATION_DELIVERED,
            :
            E_CLD_SM_ATTR_ID_CURRENT_TIER14_BLOCK16_SUMMATION_DELIVERED,

            /* Block Information Attribute set: Tier15 Block Set */
            E_CLD_SM_ATTR_ID_CURRENT_TIER15_BLOCK1_SUMMATION_DELIVERED,
            E_CLD_SM_ATTR_ID_CURRENT_TIER15_BLOCK2_SUMMATION_DELIVERED,
            :
            E_CLD_SM_ATTR_ID_CURRENT_TIER15_BLOCK16_SUMMATION_DELIVERED,

            /* Alarm Attribute set attribute ID's (D.3.2.2.9) */
            E_CLD_SM_ATTR_ID_GENERIC_ALARM_MASK = 0x0800,
            E_CLD_SM_ATTR_ID_ELECTRICITY_ALARM_MASK,
            E_CLD_SM_ATTR_ID_PRESSURE_ALARM_MASK,
            E_CLD_SM_ATTR_ID_WATER_SPECIFIC_ALARM_MASK,
            E_CLD_SM_ATTR_ID_HEAT_AND_COOLING_SPECIFIC_ALARM_MASK,
            E_CLD_SM_ATTR_ID_GAS_ALARM_MASK,
        } teCLD_SM_SimpleMeteringAttributeID;
```

8.10.2 'Meter Status' Enumerations

Enumerations for the `u8MeterStatus` element in the Simple Metering cluster structure `tsSE_SimpleMetering` are provided as #defines.

The following enumerated masks can be used to set the meter status:

Enumeration	Description
E_CLD_SM_METER_STATUS_CHECK_METER_MASK	Non-fatal problem detected on meter
E_CLD_SM_METER_STATUS_LOW_BATTERY_MASK	Battery level is low
E_CLD_SM_METER_STATUS_TAMPER_DETECT_MASK	Detected tampering with device
E_CLD_SM_METER_STATUS_POWER_FAILURE_MASK	Indicates power failure on device
E_CLD_SM_METER_STATUS_POWER_QUALITY_MASK	Power anomaly detected
E_CLD_SM_METER_STATUS_LEAK_DETECT_MASK	Detected leak (e.g. of gas or water)
E_CLD_SM_METER_STATUS_SERVICE_DISCONNECT_OPEN_MASK	Service to premises disconnected

Table 21: 'Meter Status' Enumerated Masks

8.10.3 'Unit of Measure' Enumerations

The following enumerations are used to set the `teSE_UnitOfMeasure` element in the Simple Metering cluster structure `tsSE_SimpleMetering`. Separate sets of enumerations are provided for binary and BCD (Binary Coded Decimal) representations.

```
typedef enum PACK
{
    /* Binary values */
    E_CLD_SM_UOM_KILO_WATTS                 = 0x00,
    E_CLD_SM_UOM_CUBIC_METER,
    E_CLD_SM_UOM_CUBIC_FEET,
    E_CLD_SM_UOM_100_CUBIC_FEET,            /* ccf & ccf/h */
    E_CLD_SM_UOM_US_GALLON,                 /* USG & USG/h */
    E_CLD_SM_UOM_IMPERIAL_GALLON,           /* IMPG & IMPG/h */
    E_CLD_SM_UOM_BTU,                       /* BTU & BTU/h */
    E_CLD_SM_UOM_LITERS,                    /* Liters & Liters/h */
    E_CLD_SM_UOM_KPA_GAUGE,
    E_CLD_SM_UOM_KPA_ABSOLUTE,

    /* BCD values */
    E_CLD_SM_UOM_KILO_WATTS_BCD             = 0x80,
    E_CLD_SM_UOM_CUBIC_METER_BCD,
    E_CLD_SM_UOM_CUBIC_FEET_BCD,
    E_CLD_SM_UOM_100_CUBIC_FEET_BCD,        /* ccf & ccf/h */
    E_CLD_SM_UOM_US_GALLON_BCD,             /* USG & USG/h */
    E_CLD_SM_UOM_IMPERIAL_GALLON_BCD,       /* IMPG & IMPG/h */
    E_CLD_SM_UOM_BTU_BCD,                   /* BTU & BTU/h */
    E_CLD_SM_UOM_LITERS_BCD,                /* Liters & Liters/h */
    E_CLD_SM_UOM_KPA_GAUGE_BCD,
    E_CLD_SM_UOM_KPA_ABSOLUTE_BCD
} teCLD_SM_UnitOfMeasure;
```

The above enumerations are detailed in the table below.

Enumeration	**Description**	
	Instantaneous	**Summation**
Binary Values		
E_CLD_SM_UOM_KILO_WATTS	kW (kiloWatts)	kWh (kiloWatt-hours)
E_CLD_SM_UOM_CUBIC_METER	m^3/h (cubic metres per hour)	m^3 (cubic metres)
E_CLD_SM_UOM_CUBIC_FEET	ft^3/h (cubic feet per hour)	ft^3 (cubic feet)
E_CLD_SM_UOM_100_CUBIC_FEET	ccf/h (100 cubic feet per hour)	ccf (100 cubic feet)
E_CLD_SM_UOM_US_GALLON	US gl/h (US Gallons per hour)	US gl (US Gallons)
E_CLD_SM_UOM_IMPERIAL_GALLON	Imperial gl/h (Imperial Gallons per hour)	Imperial gl (Imperial Gallons)
E_CLD_SM_UOM_BTU	BTU/h (British Thermal Units per hour)	BTU (British Thermal Units)
E_CLD_SM_UOM_LITERS	l/h (litres per hour)	l (litres)
E_CLD_SM_UOM_KPA_GAUGE	kPA (kiloPascal) gauge	-
E_CLD_SM_UOM_KPA_ABSOLUTE	kPA (kiloPascal) absolute	-
BCD Values		
E_CLD_SM_UOM_KILO_WATTS_BCD	kW (kiloWatts)	kWh (kiloWatt-hours)
E_CLD_SM_UOM_CUBIC_METER_BCD	m^3/h (cubic metres per hour)	m^3 (cubic metres)
E_CLD_SM_UOM_CUBIC_FEET_BCD	ft^3/h (cubic feet per hour)	ft^3 (cubic feet)
E_CLD_SM_UOM_100_CUBIC_FEET_BCD	ccf/h (100 cubic feet per hour)	ccf (100 cubic feet)
E_CLD_SM_UOM_US_GALLON_BCD	US gl/h (US Gallons per hour)	US gl (US Gallons)
E_CLD_SM_UOM_IMPERIAL_GALLON_BCD	Imperial gl/h (Imperial Gallons per hour)	Imperial gl (Imperial Gallons)
E_CLD_SM_UOM_BTU_BCD	BTU/h (British Thermal Units per hour)	BTU (British Thermal Units)
E_CLD_SM_UOM_LITERS_BCD	l/h (litres per hour)	l (litres)
E_CLD_SM_UOM_KPA_GAUGE_BCD	kPA (kiloPascal) gauge	-
E_CLD_SM_UOM_KPA_ABSOLUTE_BCD	kPA (kiloPascal) absolute	-

Table 22: Units of Measure Enumerations

8.10.4 'Summation Formatting' Enumerations

Enumerations for use with the `u8SummationFormatting` element in the Simple Metering cluster structure `u8SummationFormatting` are provided as #defines. The enumerations allow the following formatting information to be extracted from the `u8SummationFormatting` bitmap:

Enumeration	Description
E_CLD_SM_FORMATTING_DIGITS_TO_RIGHT_OF_DP_LS_BIT	Position of least significant bit of bit-field indicating number of digits to right of decimal point
E_CLD_SM_FORMATTING_DIGITS_TO_RIGHT_OF_DP_NUMBER_OF_BITS	Number of bits in bit-field indicating number of digits to right of decimal point
E_CLD_SM_FORMATTING_DIGITS_TO_RIGHT_OF_DP_MASK	Bit-mask used to extract number of digits to right of decimal point
E_CLD_SM_FORMATTING_DIGITS_TO_LEFT_OF_DP_LS_BIT	Position of least significant bit of bit-field indicating number of digits to left of decimal point
E_CLD_SM_FORMATTING_DIGITS_TO_LEFT_OF_DP_NUMBER_OF_BITS	Number of bits in bit-field indicating number of digits to left of decimal point
E_CLD_SM_FORMATTING_DIGITS_TO_LEFT_OF_DP_MASK	Bit-mask used to extract number of digits to left of decimal point
E_CLD_SM_FORMATTING_SUPPRESS_LEADING_ZEROS_BIT	Bit-mask used to extract bit indicating whether leading zeros will be suppressed

Table 23: 'Summation Formatting' Enumerations

The following are examples of the use of the above enumerations.

Extracting the number of digits to the right of the decimal point:

```
u8BitsToRight = (u8SummationFormatting &
E_CLD_SM_FORMATTING_DIGITS_TO_RIGHT_OF_DP_MASK)
>> E_CLD_SM_FORMATTING_DIGITS_TO_RIGHT_OF_DP_LS_BIT
```

Extracting the number of digits to the left of the decimal point:

```
u8BitsToLeft = (u8SummationFormatting &
E_CLD_SM_FORMATTING_DIGITS_TO_LEFT_OF_DP_MASK)
>> E_CLD_SM_FORMATTING_DIGITS_TO_LEFT_OF_DP_LS_BIT
```

Determining whether leading zeros will be suppressed:

```
bSuppressZeros = !((u8SummationFormatting &
E_CLD_SM_FORMATTING_SUPPRESS_LEADING_ZEROS_BIT) == 0)
```

8.10.5 ‘Supply Direction’ Enumerations

The following enumerations are used to indicate the direction of supply.

```
typedef enum PACK
{
    E_CLD_SM_CONSUMPTION_DELIVERED,
    E_CLD_SM_CONSUMPTION_RECEIVED
}teSM_IntervalChannel;
```

The above enumerations are detailed in the table below.

Enumeration	Description
E_CLD_SM_CONSUMPTION_DELIVERED	Specifies that the supply is from the customer to the utility company (in cases where the customer generates their own supply)
E_CLD_SM_CONSUMPTION_RECEIVED	Specifies that the supply is from the utility company to the customer

Table 24: ‘Supply Direction’ Enumerations

8.10.6 ‘Metering Device Type’ Enumerations

The following enumerations are used to set the `eMeteringDeviceType` element in the Simple Metering cluster structure `tsSE_SimpleMetering`.

```
typedef enum PACK
{
    E_CLD_SM_MDT_ELECTRIC                 = 0x00,
    E_CLD_SM_MDT_GAS,
    E_CLD_SM_MDT_WATER,
    E_CLD_SM_MDT_THERMAL,                 /* Deprecated */
    E_CLD_SM_MDT_PRESSURE,
    E_CLD_SM_MDT_HEAT,
    E_CLD_SM_MDT_COOLING,
    E_CLD_SM_MDT_GAS_MIRRORED             = 0x80,
    E_CLD_SM_MDT_WATER_MIRRORED,
    E_CLD_SM_MDT_THERMAL_MIRRORED,
    E_CLD_SM_MDT_PRESSURE_MIRRORED,
    E_CLD_SM_MDT_HEAT_MIRRORED,
    E_CLD_SM_MDT_COOLING_MIRRORED,
} teCLD_SM_MeteringDeviceType;
```

The above enumerations are detailed in the table below.

Enumeration	Description
E_CLD_SM_MDT_ELECTRIC	Electric Meter
E_CLD_SM_MDT_GAS	Gas Meter
E_CLD_SM_MDT_WATER	Water Meter
E_CLD_SM_MDT_THERMAL	Thermal Meter (deprecated)
E_CLD_SM_MDT_PRESSURE	Pressure Meter
E_CLD_SM_MDT_HEAT	Heat Meter
E_CLD_SM_MDT_COOLING	Cooling Meter
E_CLD_SM_MDT_GAS_MIRRORED	Mirrored Gas Meter
E_CLD_SM_MDT_WATER_MIRRORED	Mirrored Water Meter
E_CLD_SM_MDT_THERMAL_MIRRORED	Mirrored Thermal Meter (deprecated)
E_CLD_SM_MDT_PRESSURE_MIRRORED	Mirrored Pressure Meter
E_CLD_SM_MDT_HEAT_MIRRORED	Mirrored Heat Meter
E_CLD_SM_MDT_COOLING_MIRRORED	Mirrored Cooling Meter

Table 25: 'Metering Device Type' Enumerations

8.10.7 'Simple Metering Event' Enumerations

The event types generated by the Simple Metering cluster are enumerated in the `teSM_CallBackEventType` structure below:

```
typedef enum PACK
{
   E_CLD_SM_CLIENT_RECEIVED_COMMAND,
   E_CLD_SM_SERVER_RECEIVED_COMMAND,
   E_CLD_SM_FAST_POLLING_TIMER_EXPIRED
}teSM_CallBackEventType;
```

The above event types are described in the table below.

Event Type Enumeration	Description
E_CLD_SM_CLIENT_RECEIVED_COMMAND	Generated on a cluster client when a command is received from the server
E_CLD_SM_SERVER_RECEIVED_COMMAND	Generated on the cluster server when a command is received from a client
E_CLD_SM_FAST_POLLING_TIMER_EXPIRED	Generated on the cluster server when the end-time of a fast polling episode is reached (*for future use*)

Table 26: Simple Metering Event Types

8.10.8 'Server Command' Enumerations

The comands issued by a Simple Metering cluster server and received by a client are enumerated in the `teSM_ClusterServerCommands` structure below:

```
typedef enum PACK
{
    E_CLD_SM_GET_PROFILE_RESPONSE,
    E_CLD_SM_REQUEST_MIRROR,
    E_CLD_SM_REMOVE_MIRROR,
    E_CLD_SM_REQUEST_FAST_POLL_MODE_RESPONSE,
    E_CLD_SM_CLIENT_ERROR
}teSM_ClusterServerCommands;
```

Command Enumeration	Description
E_CLD_SM_GET_PROFILE_RESPONSE	Response to 'Get Profile' request - content of response is contained in the structure `tsSM_GetProfileResponseCommand` in the event (see Section 8.11.9)
E_CLD_SM_REQUEST_MIRROR	An 'Add Mirror' request
E_CLD_SM_REMOVE_MIRROR	A 'Remove Mirror' request
E_CLD_SM_REQUEST_FAST_POLL_MODE_RESPONSE	Response to 'Fast Polling' request (*for future use*)
E_CLD_SM_CLIENT_ERROR	Error condition - content of error is contained in the structure `tsSM_Error` in the event (see Section 8.11.10)

Table 27: Commands Issued by Server

8.10.9 'Client Command' Enumerations

The comands issued by a Simple Metering cluster client and received by the server are enumerated in the `teSM_ClusterClientCommands` structure below:

```
typedef enum PACK
{
    E_CLD_SM_GET_PROFILE,
    E_CLD_SM_REQUEST_MIRROR_RESPONSE,
    E_CLD_SM_MIRROR_REMOVED,
    E_CLD_SM_REQUEST_FAST_POLL_MODE,
    E_CLD_SM_SERVER_ERROR
}teSM_ClusterClientCommands;
```

Command Enumeration	Description
E_CLD_SM_GET_PROFILE	A 'Get Profile' request - content of request is contained in the structure `tsSM_GetProfileRequestCommand` in the event (see Section 8.11.8)
E_CLD_SM_REQUEST_MIRROR_RESPONSE	Response to 'Add Mirror' request - content of response is contained in the structure `tsSM_RequestMirrorResponseCommand` in the event (see Section 8.11.6)
E_CLD_SM_MIRROR_REMOVED	Response to 'Remove Mirror' request - content of response is contained in the structure `tsSM_MirrorRemovedResponseCommand` in the event (see Section 8.11.7)
E_CLD_SM_REQUEST_FAST_POLL_MODE	A 'Fast Polling' request (*for future use*)
E_CLD_SM_SERVER_ERROR	Error condition - content of error is contained in the structure `tsSM_Error` in the event (see Section 8.11.10)

Table 28: Commands Issued by Client

8.10.10 ‘Consumption Interval’ Enumerations

The following enumerations define the possible time-intervals for the consumption data captured in the ‘Get Profile’ feature.

```
typedef enum PACK
{
    E_CLD_SM_TIME_FRAME_DAILY,
    E_CLD_SM_TIME_FRAME_60MINS,
    E_CLD_SM_TIME_FRAME_30MINS,
    E_CLD_SM_TIME_FRAME_15MINS,
    E_CLD_SM_TIME_FRAME_10MINS,
    E_CLD_SM_TIME_FRAME_7_5MINS,
    E_CLD_SM_TIME_FRAME_5MINS,
    E_CLD_SM_TIME_FRAME_2_5MINS
}teSM_TimeFrame;
```

Time Frame Enumeration	Time Interval
E_CLD_SM_TIME_FRAME_DAILY	One day
E_CLD_SM_TIME_FRAME_60MINS	60 minutes
E_CLD_SM_TIME_FRAME_30MINS	30 minutes
E_CLD_SM_TIME_FRAME_15MINS	15 minutes
E_CLD_SM_TIME_FRAME_10MINS	10 minutes
E_CLD_SM_TIME_FRAME_7_5MINS	7.5 minutes
E_CLD_SM_TIME_FRAME_5MINS	5 minutes
E_CLD_SM_TIME_FRAME_2_5MINS	2.5 minutes

Table 29: ‘Consumption Interval’ Enumerations

8.10.11 ‘Simple Metering Status’ Enumerations

The following enumerations are used to report status in the Simple Metering cluster.

```
typedef enum PACK
{
    E_CLD_SM_STATUS_SUCCESS,
    E_CLD_SM_STATUS_UNDEFINED_INTERVAL_CHANNEL,
    E_CLD_SM_STATUS_INTERVAL_NOT_SUPPORTED,
    E_CLD_SM_STATUS_INVALID_END_TIME,
    E_CLD_SM_STATUS_MORE_PERIODS_REQUESTED_THAN_SUPPORTED,
    E_CLD_SM_STATUS_NO_INTERVALS_AVAILABLE_FOR_REQUESTED_TIME,
    E_CLD_SM_STATUS_EP_NOT_AVAILABLE
}teSM_Status;
```

Status Enumeration	Description
E_CLD_SM_STATUS_SUCCESS	Success
E_CLD_SM_STATUS_UNDEFINED_ INTERVAL_CHANNEL	Undefined `eIntervalChannel` value specified in ‘Get Profile’ request (see Section 8.11.8)
E_CLD_SM_STATUS_INTERVAL_ NOT_SUPPORTED	Unsupported consumption data specifed through `eIntervalChannel` in ‘Get Profile’ request (see Section 8.11.8)
E_CLD_SM_STATUS_INVALID_END_TIME	Invalid end-time specified in ‘Get Profile’ request (Section 8.11.8)
E_CLD_SM_STATUS_MORE_PERIODS_ REQUESTED_THAN_SUPPORTED	More periods specified in ‘Get Profile’ request than can be returned
E_CLD_SM_STATUS_NO_INTERVALS_ AVAILABLE_FOR_REQUESTED_TIME	No intervals available for the end-time specified in ‘Get Profile’ request
E_CLD_SM_STATUS_EP_NOT_AVAILABLE	Specified endpoint not available

Table 30: Status Enumerations

8.11 Structures

8.11.1 tsSM_CallBackMessage

For a Simple Metering event, the `eEventType` field of the `tsZCL_CallBackEvent` structure is set to E_ZCL_CBET_CLUSTER_CUSTOM. This event structure also contains an element `sClusterCustomMessage`, which is itself a structure containing a field `pvCustomData`. This field is a pointer to the following `tsSM_CallBackMessage` structure which contains the Simple Metering parameters:

```
typedef struct
{
teSM_CallBackEventType eEventType;
uint8 u8CommandId;

    union
    {
        tsSM_GetProfileResponseCommand       sGetProfileResponseCommand;
        tsSM_RequestFastPollResponseCommand  sRequestFastPollResponseCommand;
        tsSM_GetProfileRequestCommand        sGetProfileCommand;
        tsSM_RequestMirrorResponseCommand    sRequestMirrorResponseCommand;
        tsSM_MirrorRemovedResponseCommand    sMirrorRemovedResponseCommand;
        tsSM_RequestFastPollCommand          sRequestFastPollCommand;
        tsSM_Error                           sError;
    }uMessage;

}tsSM_CallBackMessage;
```

where:

- `eEventType` is the Simple Metering event type from those listed in Section 8.10.7
- `u8CommandId` is the identifier of the type of Simple Metering command received. This field is only valid for the following Simple Metering event types:
  - E_CLD_SM_CLIENT_RECEIVED_COMMAND - enumerated commands are provided, as described in Section 8.10.8
  - E_CLD_SM_SERVER_RECEIVED_COMMAND - enumerated commands are provided, as described in Section 8.10.9
- `uMessage` is a union containing the command payload in one of the following forms (depending on the command specified in the field `u8CommandId`):
  - `sGetProfileResponseCommand` is a structure containing the payload of a ‘Get Profile’ response - see Section 8.11.9
  - `sRequestFastPollResponseCommand` is a structure containing the payload of a ‘Fast Polling’ response (*for future use*)
  - `sGetProfileCommand` is a structure containing the payload of a ‘Get Profile’ request - see Section 8.11.8
  - `sRequestMirrorResponseCommand` is a structure containing the payload of an ‘Add Mirror’ response - see Section 8.11.6

- `sMirrorRemovedResponseCommand` is a structure containing the payload of an ‘Remove Mirror’ response - see Section 8.11.7
- `sRequestFastPollCommand` is a structure containing the payload of an ‘Fast Polling’ request (*for future use*)
- `sError` is a structure containing the details of an error condition - see Section 8.11.10

8.11.2 tsSE_Mirror

Details of the mirror endpoints on the ESP are kept in an array of structures of the type `tsSE_Mirror` (one structure per endpoint) within the `tsSE_EspMeterDevice` structure (see Section 13.2.1). The `tsSE_Mirror` structure is shown and described below.

Note: This structure is only for use by the profile software and should not be modified by the application.

```
typedef struct
{
    /*Mirrored EndPoint*/
    tsZCL_EndPointDefinition sEndPoint;

    /*Mirror Requester address*/
    uint64u64SourceAddress;

    /*Mirror cluster instances*/
    tsSE_MirrorClusterInstances sSEMirrorClusterInstances;

    /*Event Address, Custom callback event, Custom callback message*/
    tsSM_CustomStruct sSMMirrorCustomDataStruct;

}tsSE_Mirror;
```

where:

- `sEndPoint` is a `tsZCL_EndPointDefinition` structure which contains details of the endpoint corresponding to the mirror (for a description of this structure, refer to the *ZCL User Guide (JN-UG-3077)*)
- `u64SourceAddress` is the 64-bit IEEE address of the Metering Device to which the mirror endpoint is assigned - a zero value indicates that the mirror endpoint is not currently assigned to a device

- `sSEMirrorClusterInstances` is a `tsSE_MirrorClusterInstances` structure (see Section 8.11.3) which contains information on the Basic and Simple Metering cluster instances that are associated with the mirror endpoint
- `sSMMirrorCustomDataStruct` is a `tsSM_CustomStruct` structure (see Section 8.11.4) which contains data relating to a received command/message for the mirror endpoint

8.11.3 tsSE_MirrorClusterInstances

This structure contains information on the Basic and Simple Metering cluster instances that are associated with a mirror endpoint.

Note: This structure is only for use by the profile software and should not be modified by the application.

```
typedef struct
{
   /*Basic Cluster Instance*/
   tsZCL_ClusterInstance sBasicCluster;

   /* SM Cluster Instance */
   tsZCL_ClusterInstance sSM_Cluster;

}tsSE_MirrorClusterInstances;
```

where:

- `sBasicCluster` is a `tsZCL_ClusterInstance` structure which contains information on the Basic cluster instance associated with a mirror endpoint (for a description of this structure, refer to the *ZCL User Guide (JN-UG-3077)*)
- `sSM_Cluster` is a `tsZCL_ClusterInstance` structure which contains information on the Simple Metering cluster instance associated with a mirror endpoint (for a description of this structure, refer to the *ZCL User Guide (JN-UG-3077)*)

8.11.4 tsSM_CustomStruct

This structure contains data relating to a command/message for a mirror endpoint.

Note: This structure is only for use by the profile software and should not be modified by the application.

```
typedef struct
{
    tsZCL_ReceiveEventAddress  sReceiveEventAddress;
    tsZCL_CallBackEvent        sSMCustomCallBackEvent;
    tsSM_CallBackMessage       sSMCallBackMessage;
} tsSM_CustomStruct;
```

where:

- `sReceiveEventAddress` is a `tsZCL_ReceiveEventAddress` structure which contains addressing information relating to a received mirroring command/message (for a description of this structure, refer to the *ZCL User Guide (JN-UG-3077)*)
- `sSMCustomCallBackEvent` is a `tsZCL_CallBackEvent` structure which contains the event that has been generated as a result of the received command/message (for a description of this structure, refer to the *ZCL User Guide (JN-UG-3077)*)
- `sSMCallBackMessage` is a `tsSM_CallBackMessage` structure (see Section 8.11.1) which contains details of the event and the command/message that caused the event

8.11.5 tsSEGetProfile

This structure is used to store historical consumption data when the 'Get Profile' feature is enabled. The data within the structure corresponds to a single consumption interval.

```
typedef struct
{
   uint32 u32UtcTime;
   zuint24 u24ConsumptionReceived;
   zuint24 u24ConsumptionDelivered;
}tsSEGetProfile;
```

where:

- `u32UtcTime` is the end-time of the consumption interval (as a UTC time)
- `u24ConsumptionReceived` is the number of units received from the customer during the interval (for customers who generate and sell their own units)
- `u24ConsumptionDelivered` is the number of units delivered to the customer during the interval

8.11.6 tsSM_RequestMirrorResponseCommand

This structure contains the details of an 'Add Mirror' response (from a cluster client). It is included in the structure `tsSM_CallBackMessage` when an E_CLD_SM_SERVER_RECEIVED_COMMAND event containing the command E_CLD_SM_REQUEST_MIRROR_RESPONSE is generated on the cluster server.

```
typedef struct
{
    uint16 u16Endpoint;
}tsSM_RequestMirrorResponseCommand;
```

where `u16Endpoint` is the number of the endpoint on which the mirror was successfully added or takes the value 0xFFFF if the request failed because no free endpoint was available for the mirror.

8.11.7 tsSM_MirrorRemovedResponseCommand

This structure contains the details of a 'Remove Mirror' response (from a cluster client). It is included in the structure `tsSM_CallBackMessage` when an E_CLD_SM_SERVER_RECEIVED_COMMAND event containing the command E_CLD_SM_MIRROR_REMOVED is generated on the cluster server.

```
typedef struct
{
    uint16 u16Endpoint;
}tsSM_MirrorRemovedResponseCommand;
```

where `u16Endpoint` is the number of the endpoint from which the mirror was successfully removed, or takes the value 0xFFFF if the remove request failed.

8.11.8 tsSM_GetProfileRequestCommand

This stucture contains the details of a 'Get Profile' request (from a cluster client). It is included in the structure `tsSM_CallBackMessage` when an E_CLD_SM_SERVER_RECEIVED_COMMAND event containing the command E_CLD_SM_GET_PROFILE is generated on the cluster server.

```
typedef struct
{
    teSM_IntervalChannel eIntervalChannel;
    uint8 u8NumberOfPeriods;
    uint8 u8SourceEndPoint;
    uint8 u8DestinationEndPoint;
    uint32 u32EndTime;
    tsZCL_Address sSourceAddress;
}tsSM_GetProfileRequestCommand;
```

where:

- `eIntervalChannel` is a value indicating the required consumption data:
  - E_CLD_SM_CONSUMPTION_RECEIVED - units from customer
  - E_CLD_SM_CONSUMPTION_DELIVERED - units to customer
- `u8NumberOfPeriods` is the number of consumption intervals for which data is being requested
- `u8SourceEndPoint` is the number of the source endpoint of the request on the client
- `u8DestinationEndPoint` is the number of the destination endpoint of the request on the server
- `u32EndTime` is the end-time for which consumption data is being requested - the most recent consumption data will be reported which has an end-time equal

to or earlier than this end-time (a zero value will result in the most recent consumption data)

- `sSourceAddress` is a structure containing the source address of the request - that is, the address of the requesting client (the structure is described in the *ZCL User Guide (JN-UG-3077)*)

8.11.9 tsSM_GetProfileResponseCommand

This stucture contains the details of a 'Get Profile' response (from the cluster server). It is included in the structure `tsSM_CallBackMessage` when an E_CLD_SM_CLIENT_RECEIVED_COMMAND event containing the command E_CLD_SM_GET_PROFILE_RESPONSE is generated on the cluster server.

```
typedef struct
{
    uint32         u32Endtime;
    teSM_Status    eStatus;
    teSM_TimeFrame u8ProfileIntervalPeriod;
    uint8          u8NumberOfPeriodsDelivered;
    zuint24       *pau24Intervals;
}tsSM_GetProfileResponseCommand;
```

where:

- `u32Endtime` is the end-time of the consumption data that is being reported, as a UTC time
- `eStatus` is the status of the response, represented by one of the enumerated values listed in Section 8.10.11
- `u8ProfileIntervalPeriod` is the time-interval (consumption interval) over which each set of consumption data is collected - one of the standard enumerated values listed in Section 8.10.10
- `u8NumberOfPeriodsDelivered` is the number of consumption intervals being reported
- `pau24Intervals` is a pointer to the consumption data being reported

8.11.10 tsSM_Error

This stucture contains the details of an error response (from cluster server or client). It is included in the structure `tsSM_CallBackMessage` when an E_CLD_SM_SERVER_RECEIVED_COMMAND event is generated containing the command E_CLD_SM_SERVER_ERROR on a client or E_CLD_SM_CLIENT_ERROR on the server.

```
typedef struct
{
    uint8 u8Endpoint;
    uint8 u8Status;
}tsSM_Error;
```

where

- `u8Endpoint` is the number of the endpoint from which the error is reported
- `u8Status` is a value representing the nature of the error

8.12 Compile-Time Options

This section describes the compile-time options that may be enabled in the **zcl_options.h** file of an application that uses the Simple Metering cluster.

The Simple Metering cluster is enabled by defining CLD_SIMPLE_METERING.

Optional Attributes

The optional attributes for the Simple Metering cluster are enabled/disabled by defining:

- For optional attributes from ‘Reading Information’ attribute set:
  - CLD_SM_ATTR_CURRENT_SUMMATION_RECEIVED
  - CLD_SM_ATTR_CURRENT_MAX_DEMAND_DELIVERED
  - CLD_SM_ATTR_CURRENT_MAX_DEMAND_RECEIVED
  - CLD_SM_ATTR_DFT_SUMMATION
  - CLD_SM_ATTR_DAILY_FREEZE_TIME
  - CLD_SM_ATTR_POWER_FACTOR
  - CLD_SM_ATTR_READING_SNAPSHOT_TIME
  - CLD_SM_ATTR_CURRENT_MAX_DEMAND_DELIVERED_TIME
  - CLD_SM_ATTR_CURRENT_MAX_DEMAND_RECEIVED_TIME
- For optional attributes from ‘Time-Of-Use (TOU) Information’ attribute set:
  - CLD_SM_ATTR_CURRENT_TIER_1_SUMMATION_DELIVERED
  - CLD_SM_ATTR_CURRENT_TIER_1_SUMMATION_RECEIVED
  - CLD_SM_ATTR_CURRENT_TIER_2_SUMMATION_DELIVERED
  - CLD_SM_ATTR_CURRENT_TIER_2_SUMMATION_RECEIVED
  - CLD_SM_ATTR_CURRENT_TIER_3_SUMMATION_DELIVERED
  - CLD_SM_ATTR_CURRENT_TIER_3_SUMMATION_RECEIVED
  - CLD_SM_ATTR_CURRENT_TIER_4_SUMMATION_DELIVERED
  - CLD_SM_ATTR_CURRENT_TIER_4_SUMMATION_RECEIVED
  - CLD_SM_ATTR_CURRENT_TIER_5_SUMMATION_DELIVERED
  - CLD_SM_ATTR_CURRENT_TIER_5_SUMMATION_RECEIVED
  - CLD_SM_ATTR_CURRENT_TIER_6_SUMMATION_DELIVERED
  - CLD_SM_ATTR_CURRENT_TIER_6_SUMMATION_RECEIVED
- For optional attributes from ‘Block Information’ attribute set:
  - CLD_SM_ATTR_NO_TIER_BLOCK_CURRENT_SUMMATION_DELIVERED_MAX_COUNT (maximum value of 16)
  - CLD_SM_ATTR_NUM_OF_TIERS_CURRENT_SUMMATION_DELIVERED (maximum value of 15)
  - CLD_SM_ATTR_NUM_OF_BLOCKS_IN_EACH_TIER_CURRENT_SUMMATION_DELIVERED (maximum value of 16)
- For optional attributes from ‘Formatting’ attribute set:
  - CLD_SM_ATTR_MULTIPLIER
  - CLD_SM_ATTR_DIVISOR
  - CLD_SM_ATTR_DEMAND_FORMATING
  - CLD_SM_ATTR_HISTORICAL_CONSUMPTION_FORMATTING

- For optional attributes from 'ESP Historical Consumption' attribute set:
  - CLD_SM_ATTR_INSTANTANEOUS_DEMAND
  - CLD_SM_ATTR_CURRENT_DAY_CONSUMPTION_DELIVERED
  - CLD_SM_ATTR_CURRENT_DAY_CONSUMPTION_RECEIVED
  - CLD_SM_ATTR_PREVIOUS_DAY_CONSUMPTION_DELIVERED
  - CLD_SM_ATTR_PREVIOUS_DAY_CONSUMPTION_RECEIVED
  - CLD_SM_ATTR_CURRENT_PARTIAL_PROFILE_INTERVAL_START_TIME_DELIVERED
  - CLD_SM_ATTR_CURRENT_PARTIAL_PROFILE_INTERVAL_START_TIME_RECEIVED
  - CLD_SM_ATTR_CURRENT_PARTIAL_PROFILE_INTERVAL_VALUE_DELIVERED
  - CLD_SM_ATTR_CURRENT_PARTIAL_PROFILE_INTERVAL_VALUE_RECEIVED
- For optional attribute from 'Load Profile' attribute set:
  - CLD_SM_ATTR_MAX_NUMBER_OF_PERIODS_DELIVERED
- For optional attributes from 'Supply Limit' attribute set:
  - CLD_SM_ATTR_CURRENT_DEMAND_DELIVERED
  - CLD_SM_ATTR_DEMAND_LIMIT
  - CLD_SM_ATTR_DEMAND_INTEGRATION_PERIOD
  - CLD_SM_ATTR_NUMBER_OF_DEMAND_SUBINTERVALS

Mirroring

If the mirroring of metering data is to be enabled (see Section 8.5), the following options must be defined in the **zcl_options.h** file.

On the Simple Metering server on the Metering Device (which will request and report to a mirror on a mirroring server, such as the ESP), there is no need to define anything.

On the Simple Metering client on the mirroring server, such as the ESP, the mirroring option must be enabled by including:

```
#define CLD_SM_SUPPORT_MIRROR
```

In addition, the following defines must be added on the mirroring server (e.g. ESP):

```
#define CLD_BAS_ATTR_PHYSICAL_ENVIRONMENT
```

(flags support for mirroring via a non-zero value of the `u8PhysicalEnvironment` attribute of the Basic cluster)

```
#define CLD_SM_NUMBER_OF_MIRRORS      <n>
```

(sets the maximum number of mirrors supported on the mirroring server to the value n)

```
#define ZCL_ATTRIBUTE_REPORTING_CLIENT_SUPPORTED
```

(enables support for attribute reporting clients)

The Simple Metering cluster attributes that will be supported by mirroring must be defined on the mirroring server (the same set of attributes are mirrored on all endpoints):

- CLD_SM_MIRROR_ATTR_CURRENT_SUMMATION_RECEIVED
- CLD_SM_MIRROR_ATTR_CURRENT_MAX_DEMAND_DELIVERED
- CLD_SM_MIRROR_ATTR_CURRENT_MAX_DEMAND_RECEIVED
- CLD_SM_MIRROR_ATTR_DFT_SUMMATION
- CLD_SM_MIRROR_ATTR_DAILY_FREEZE_TIME
- CLD_SM_MIRROR_ATTR_POWER_FACTOR
- CLD_SM_MIRROR_ATTR_READING_SNAPSHOT_TIME
- CLD_SM_MIRROR_ATTR_CURRENT_MAX_DEMAND_DELIVERED_TIME
- CLD_SM_MIRROR_ATTR_CURRENT_MAX_DEMAND_RECEIVED_TIME
- CLD_SM_MIRROR_ATTR_CURRENT_TIER_1_SUMMATION_DELIVERED
- CLD_SM_MIRROR_ATTR_CURRENT_TIER_1_SUMMATION_RECEIVED
- CLD_SM_MIRROR_ATTR_CURRENT_TIER_2_SUMMATION_DELIVERED
- CLD_SM_MIRROR_ATTR_CURRENT_TIER_2_SUMMATION_RECEIVED
- CLD_SM_MIRROR_ATTR_CURRENT_TIER_3_SUMMATION_DELIVERED
- CLD_SM_MIRROR_ATTR_CURRENT_TIER_3_SUMMATION_RECEIVED
- CLD_SM_MIRROR_ATTR_CURRENT_TIER_4_SUMMATION_DELIVERED
- CLD_SM_MIRROR_ATTR_CURRENT_TIER_4_SUMMATION_RECEIVED
- CLD_SM_MIRROR_ATTR_CURRENT_TIER_5_SUMMATION_DELIVERED
- CLD_SM_MIRROR_ATTR_CURRENT_TIER_5_SUMMATION_RECEIVED
- CLD_SM_MIRROR_ATTR_CURRENT_TIER_6_SUMMATION_DELIVERED
- CLD_SM_MIRROR_ATTR_CURRENT_TIER_6_SUMMATION_RECEIVED
- CLD_SM_MIRROR_ATTR_MULTIPLIER
- CLD_SM_MIRROR_ATTR_DIVISOR
- CLD_SM_MIRROR_ATTR_DEMAND_FORMATING
- CLD_SM_MIRROR_ATTR_HISTORICAL_CONSUMPTION_FORMATTING
- CLD_SM_MIRROR_ATTR_INSTANTANEOUS_DEMAND
- CLD_SM_MIRROR_ATTR_CURRENT_DAY_CONSUMPTION_DELIVERED
- CLD_SM_MIRROR_ATTR_CURRENT_DAY_CONSUMPTION_RECEIVED
- CLD_SM_MIRROR_ATTR_PREVIOUS_DAY_CONSUMPTION_DELIVERED
- CLD_SM_MIRROR_ATTR_PREVIOUS_DAY_CONSUMPTION_RECEIVED
- CLD_SM_MIRROR_ATTR_CURRENT_PARTIAL_PROFILE_INTERVAL_START_TIME_DELIVERED
- CLD_SM_MIRROR_ATTR_CURRENT_PARTIAL_PROFILE_INTERVAL_START_TIME_RECEIVED
- CLD_SM_MIRROR_ATTR_CURRENT_PARTIAL_PROFILE_INTERVAL_VALUE_DELIVERED
- CLD_SM_MIRROR_ATTR_CURRENT_PARTIAL_PROFILE_INTERVAL_VALUE_RECEIVED
- CLD_SM_MIRROR_ATTR_MAX_NUMBER_OF_PERIODS_DELIVERED
- CLD_SM_MIRROR_ATTR_CURRENT_DEMAND_DELIVERED
- CLD_SM_MIRROR_ATTR_DEMAND_LIMIT
- CLD_SM_MIRROR_ATTR_DEMAND_INTEGRATION_PERIOD
- CLD_SM_MIRROR_ATTR_NUMBER_OF_DEMAND_SUBINTERVALS

The Basic cluster attributes that will be supported by mirroring must also be defined on the mirroring server (the same set of attributes are mirrored on all endpoints), from the following:

- CLD_BAS_MIRROR_ATTR_APPLICATION_VERSION
- CLD_BAS_MIRROR_ATTR_STACK_VERSION
- CLD_BAS_MIRROR_ATTR_HARDWARE_VERSION
- CLD_BAS_MIRROR_ATTR_MANUFACTURER_NAME
- CLD_BAS_MIRROR_ATTR_MODEL_IDENTIFIER
- CLD_BAS_MIRROR_ATTR_DATE_CODE
- CLD_BAS_MIRROR_ATTR_LOCATION_DESCRIPTION
- CLD_BAS_MIRROR_ATTR_PHYSICAL_ENVIRONMENT
- CLD_BAS_MIRROR_ATTR_DEVICE_ENABLED
- CLD_BAS_MIRROR_ATTR_ALARM_MASK
- CLD_BAS_MIRROR_ATTR_DISABLE_LOCAL_CONFIG

Get Profile

If the ‘Get Profile’ feature is to be used (see Section 8.6), the following options must be defined in the **zcl_options.h** file.

The ‘Get Profile’ option must be enabled on the server and clients by including:

```
#define CLD_SM_SUPPORT_GET_PROFILE
```

Then, the following must be included on the server (only):

```
#ifdef CLD_SM_SUPPORT_GET_PROFILE
    #defineCLD_SM_GETPROFILE_MAX_NO_INTERVALS   <n>
#endif
```

where <n> is the maximum number of consumption intervals to be held on the server (and therefore determines the amount of memory to be reserved for the circular buffer that is used to store the data for these consumption intervals).

9. Demand-Response and Load Control Cluster

This chapter outlines the Demand-Response and Load Control (DRLC) cluster which is defined in the ZigBee Smart Energy profile. The cluster is able to receive load control requests from the utility company and act upon them by controlling an attached appliance, such as a heater or pump - this is the 'demand-response' functionality.

The DRLC cluster has a Cluster ID of 0x0701.

9.1 Overview

The DRLC cluster is required in SE devices as indicated in the table below.

	Server-side	**Client-side**
Mandatory in...	ESP	PCT
		Load Control Device
Optional in...		IPD
		Smart Appliance

Table 31: DRLC Cluster in SE Devices

The ESP acts as the DRLC cluster server, since it is the device which receives Load Control Events (LCEs) from the utility company via the backhaul network. Other devices act as clients and receive the LCEs forwarded by the ESP:

- An IPD would normally display a list of LCEs to allow the consumer to manually modify consumption.
- A Load Control Device, PCT or Smart Appliance would participate in an LCE by automatically adjusting the consumption of the device.

Devices that participate in an LCE must report their participation back to the ESP. Participation may result in the consumer receiving a credit on their utility bill.

Note: In the current NXP implementation of ZigBee SE, the DRLC cluster client is contained within an IPD only. This illustrates how to incorporate the DRLC cluster in other devices which need to participate in LCEs.

The LCEs contain a time-stamp. Therefore, devices which support the DRLC cluster client and which participate in LCEs must implement the Time cluster and maintain a real-time clock.

The DRLC cluster is enabled by defining CLD_DRLC in the **zcl_options.h** file - see Section 3.5.1. Further compile-time options for the DRLC cluster are detailed in Section 9.12.

9.2 DRLC Cluster Structure and Attributes

The DRLC cluster has no server attributes but has client attributes that are contained in the following `tsCLD_DRLC` structure:

```
typedef struct
{
    uint8                   u8UtilityEnrolmentGroup;
    uint8                   u8StartRandomizeMinutes;
    uint8                   u8StopRandomizeMinutes;
    uint16                  u16DeviceClassValue;
} tsCLD_DRLC;
```

where:

- `u8UtilityEnrolmentGroup` identifies the 'enrolment' group to which the device belongs, where a group of devices is defined by the utility company in order to aid load management in a large system. The default value of 0x00 is used to indicate membership of all groups.
- `u8StartRandomizeMinutes` specifies the largest random delay, in minutes, that can be applied to the start of a Load Control Event (so a random delay, no greater than this value, will be applied to an individual event). The valid range of values is 0x00 to 0x3C (0 to 60 mins), where 0x00 indicates that no delay is to be applied.
- `u8StopRandomizeMinutes` specifies the largest random delay, in minutes, that can be applied to the end of a Load Control Event (so a random delay, no greater than this value, will be applied to an individual event). The valid range of values is 0x00 to 0x3C, where 0x00 indicates that no delay is to be applied.
- `u16DeviceClassValue` is a bitmap specifying the relevant device classes (e.g. water heater and pool pump). Enumerations are provided for the device classes and are detailed in Section 9.10.1. If more than one device class is required, the relevant enumerations can be bitwise-ORed.

Note: It may be desirable to refuse write access to the `u16DeviceClassValue` attribute on a device. To do this, when a 'write attributes' request is received and an E_ZCL_CBET_CHECK_ATTRIBUTE_RANGE event is generated for this attribute, the application should set the `eAttributeStatus` field of the event to E_ZCL_DENY_ATTRIBUTE_ACCESS.

9.3 Initialisation

Provided that the DRLC cluster is enabled in the compile-time options (see Section 9.12), the cluster will be automatically initialised when the SE profile is initialised and the SE device is registered in the application - that is, by calling **eSE_Initialise()** and the relevant endpoint registration function for the device, for example:

- **eSE_RegisterEspEndPoint()** on a standalone ESP (cluster server)
- **eSE_RegisterIPDEndPoint()** on an IPD (cluster client)

As part of this initialisation, the DRLC cluster is created and, on the ESP, a DRLC timer server is registered to support time-stamps in the LCEs.

A DRLC cluster client must also perform a number of other initialisation steps in order to establish communication with the cluster server. These are described below.

1. **Set 'device class' attribute:** The value of the 'device class' attribute (see Section 9.2) must be set immediately after **eSE_RegisterIPDEndPoint()** is called and before the network is started.
2. **Bind to server:** A non-sleeping client should bind its endpoint to the server using the ZigBee PRO API function **ZPS_eAplZdpBindUnbindRequest()**. This allows the server to send out unsolicited LCEs to the client.

   Before this binding can take place, the client must obtain the IEEE/MAC address of the ESP/server. This can be achieved by first using the function **ZPS_eAplZdpMatchDescRequest()** to find the ESP/Server and to obtain its network address. The function **ZPS_eAplZdpIeeeAddrRequest()** can then be used to obtain the corresponding IEEE/MAC address. Once found, both addresses must be added to the local Address Map using the function **ZPS_eAplZdoAddAddrMapEntry()**.

   All four of the above ZPS functions are described in the *ZigBee PRO Stack User Guide (JN-UG-3048)*.
3. **Synchronise time with ESP:** A client should synchronise ZCL time with the ESP using the Time cluster as soon as initialisation is complete. It is not possible to process unsolicited LCEs with a 'start-time of now' until ZCL time has been synchronised.

Once the clients have been set up, the ESP/server may need to configure the enrolment groups and randomisation attributes of the DRLC clients (see Section 9.2). The ESP may use one of the following mechanisms to determine when a DRLC client has come on-line:

- A Get Scheduled Events message is received from a new client
- A Report Event Status message is received from a new client
- A binding request is received from a new client

9.4 Load Control Events (LCEs)

A Load Control Event (LCE) is an instruction, which originates from the utility company, to schedule a temporary adjustment of consumption in devices that support the DRLC cluster. The contents of an LCE are outlined in Section 9.4.1.

An LCE is sent from the utility company to the DRLC server (ESP) of an SE network, from where it is is passed to DRLC clients. The LCEs are held in lists on the server and clients, as described in Section 9.4.2.

LCE handling is described in Section 9.5.

9.4.1 LCE Contents

The information contained in an LCE includes:

- LCE ID (provided by utility company)
- Target device class and enrolment group
- Start-time
- Duration
- Criticality level
- Required adjustment(s)
- Randomisation requirements (for start-time and end-time)

For a full list and description of the LCE data, refer to the description of the LCE structure `tsSE_DRLCLoadControlEvent` in Section 9.11.1.

9.4.2 LCE Lists

The DRLC cluster server and clients each hold the following lists of LCEs:

- **Active list:** Contains LCEs that are currently being executed - it is possible for more than one LCE to be active at the same time, provided that their device classes and enrolment groups do not clash.
- **Scheduled list:** Contains LCEs that are due to be executed in the future - that is, their start-time is later than the current time.
- **Cancelled list:** Contains LCEs that have been cancelled with a randomised end-time and whose random end-time has not yet been reached.
- **Deallocated list:** Contains expired LCEs and therefore a record of the free storage for LCEs - used internally by the cluster (and not by the application).

A new LCE is first added to the Scheduled list, unless it has a ‘start-time of now’ in which case it is added to the Active list. An LCE in the Scheduled list is automatically moved to the Active list at the scheduled start-time (or at the randomised start-time). At the end of an active LCE, it is automatically moved to the Deallocated list. However, an active LCE which is cancelled with a randomised end-time is automatically moved to the Cancelled list, where it stays until the end-time has been reached (when it is moved to the Deallocated list).

The addition of a new LCE on the cluster server is performed by the server application, as described in Section 9.5.1, but is done automatically by the cluster on the clients. All other operations on LCE lists, apart from cancellation (see Section 9.5.3), are performed automatically by the cluster on both server and client.

Functions are provided to access entries in the local LCE lists:

- **eSE_DRLCGetLoadControlEvent()** can be used to obtain a particular LCE entry (with specified list index) in any one of the local lists
- **eSE_DRLCFindLoadControlEvent()** can be used to search for and obtain a particular LCE (with specified ID) in any of the local lists

9.5 LCE Handling

LCEs are handled on the DRLC cluster server and clients as described in Section 9.5.1 and Section 9.5.2 respectively. Cancelling LCEs is described in Section 9.5.3.

Note: The DRLC callback events referred to in this section are further described in Section 9.7 and are handled by the callback function that is registered as part of the device endpoint registration.

9.5.1 LCE Handling on Server

When a new LCE is received from the utility company, it is the responsibility of the application on the ESP (DRLC cluster server) to add this LCE to the local ‘Scheduled’ list (or to the ‘Active’ list, if the LCE has a ‘start-time of now’). This addition is performed using the function **eSE_DRLCAddLoadControlEvent()**, which also sends the LCE (unsolicited) to the cluster clients. The LCE should normally be sent to all client endpoints with which the cluster server has been bound (see Section 9.3).

Note 1: Following the initial reception of LCEs from the utility company, the addition of these LCEs to the list(s) through **eSE_DRLCAddLoadControlEvent()** can be done after calling **eSE_RegisterEspMeterEndPoint()** or **eSE_RegisterEspEndPoint()** but before calling **ZPS_eAplZdoStartStack()**.

Note 2: On receiving an LCE, the client will check the device class and enrolment group specified within the LCE, and will only accept the LCE if these values match the corresponding DRLC cluster attributes held locally (see Section 9.2).

The cluster server also automatically responds to Get Scheduled Events messages from cluster clients that need current and future LCEs (see Section 9.5.2.2).

9.5.2 LCE Handling on Clients

The sub-sections below describe the various LCE handling activities that take place on a DRLC cluster client.

9.5.2.1 LCE Activation and De-activation

On receiving a new LCE from the DRLC cluster server, a cluster client first checks the device class and enrolment group specified within the LCE. If they do not match those of the local device (see DRLC attributes in Section 9.2), the LCE is discarded.

Note: A DRLC cluster client can opt out of an individual LCE using the **eSE_DRLCSetEventUserOption()** function.

Generally, a valid LCE received from the cluster server is automatically added to the 'Scheduled' list on the client - the E_SE_DRLC_EVENT_COMMAND callback event containing the command SE_DRLC_LOAD_CONTROL_EVENT is generated on the client to indicate that this has been done. However, if the LCE has a 'start-time of now', it is added directly to the 'Active' list, provided that the start-time is not randomised (see below).

If a new LCE is successfully added to the Scheduled (or Active) list, the client will send a Report Event Status message to the server to confirm acceptance of the LCE.

When the start-time of an LCE in the 'Scheduled' list is reached, the LCE is automatically moved to the 'Active' list - the E_SE_DRLC_EVENT_ACTIVE callback event is generated on the client to indicate that this has been done, allowing the application to make the required load adjustment. However, if a randomised start-time is enabled (in the LCE), the move to the 'Active' list is delayed by a random time interval that is no greater than the maximum defined by the cluster attribute `u8StartRandomizeMinutes` (see Section 9.2).

When the duration of the active LCE has expired, the LCE is automatically moved to the 'De-allocated' list - the E_SE_DRLC_EVENT_EXPIRED callback event is generated on the client to indicate that this has been done, allowing the application to restore the load to the previous level. However, if a randomised end-time is enabled (in the LCE), the move to the 'Deallocated' list is delayed by a random time interval that is no greater than the maximum defined by the cluster attribute `u8StopRandomizeMinutes` (see Section 9.2).

Note: The above randomise attributes of the DRLC cluster also allow LCE start-time and end-time randomisation to be disabled for all LCEs on the local device. If this is the case, randomisation settings within the LCE itself will be ignored.

9.5.2.2 Getting Scheduled Events

The application on the DRLC cluster client can send a Get Scheduled Events message to the cluster server in order to obtain relevant current and future LCEs. This message may be used in the following situations:

- On a non-sleeping device, the application may send this message:
  - immediately after binding with the cluster server in order to get the initial LCEs (subsequent LCEs will be received unsolicited from the server)
  - at other times in order to top up its LCE list, if it has previously discarded an LCE due to lack of storage
- On a sleeping device (End Device), the application may send this message on waking from sleep in order to obtain new LCEs that were distributed by the cluster server during sleep (and therefore not received).

The Get Scheduled Events message can be sent from a client using the function **eSE_DRLCGetScheduledEventsSend()**. The message includes the earliest start-time of the LCEs of interest, where zero is used to indicate all LCEs - for a sleeping End Device, this time should be set to zero or the current time, in case there are replacements on the server for LCEs already in the client's lists. The message also allows the maximum number of returned LCEs to be specified, where zero is used to indicate all LCEs.

Note: The arrival of the Get Scheduled Events message will result in the generation of the E_SE_DRLC_EVENT_COMMAND callback event, containing a DRLC_GET_SCHEDULED_EVENTS command, on the cluster server. However, the cluster will respond to the message automatically.

On receiving the requested LCEs from the cluster server, the cluster client automatically updates the local LCE lists with the reported LCEs.

9.5.2.3 Reporting LCE Actions to Server

By default, a DRLC cluster client sends a Report Event Status message to the cluster server when an LCE is actioned on the client - that is, when an LCE is moved between lists on the client, such as from 'Scheduled' to 'Active' or from 'Active' to 'Deallocated' (see Section 9.4.2). Details of the actioned LCE are sent in a `tsSE_DRLCReportEvent` structure (see Section 9.11.4). The nature of the action is indicated in this stucture using an enumeration (see Section 9.10.8).

Note: The DRLC cluster server is informed of the arrival of a Report Event Status message via the callback event E_SE_DRLC_EVENT_COMMAND, containing a SE_DRLC_REPORT_EVENT_STATUS command. The ESP/server may inform the utility company of the reported status - if the message cannot be forwarded immediately then it must be buffered by the application.

If a DRLC cluster client opts out of a particular LCE using the function **eSE_DRLCSetEventUserOption()**, a Report Event Status message is sent to the cluster server to indicate this. On reaching the end-time of the LCE, another Report Event Status message is sent to the server to confirm that the LCE has completed without the participation of the local client.

9.5.2.4 Over-riding LCE Settings

The client application can over-ride certain aspects of an LCE using the function **eSE_DRLCSetEventUserData()**, which allows load control data values to be modified, including:

- Criticality level
- Cooling temperature set-point
- Heating temperature set-point
- Load adjustment percentage
- Duty cycle

For example, the ESP/server may request an HVAC device to set its cooling level to 24^{o}C, but the user may choose to over-ride this with a cooling level of 20^{o}C. The above data values and their formats are detailed in the LCE structure description in Section 9.11.1.

The function **eSE_DRLCSetEventUserData()** modifies one load control data value on each call. Therefore, in order to modify more than one data value, the function must be called multiple times.

When a change is made, the cluster client automatically notifies the cluster server by sending a Report Event Status message containing the change.

9.5.3 Cancelling LCEs

An LCE can be cancelled, in which case it is moved to the 'Deallocated' list (possibly via the 'Cancelled' list - see below). A cancellation can only be performed from the DRLC cluster server and is normally sent to all client endpoints that have been bound to the server. Two functions are provided which can be called on the cluster server to perform LCE cancellations:

- **eSE_DRLCCancelLoadControlEvent()** is used to cancel a particular LCE
- **eSE_DRLCCancelAllLoadControlEvents()** is used to cancel all LCEs

Cancellation involves removing the LCE(s) from the 'Scheduled' or 'Active' lists on the cluster server and clients, which is done automatically by the cluster. As a result, the callback event E_SE_DRLC_EVENT_COMMAND is generated, containing a LOAD_CONTROL_EVENT_CANCEL or LOAD_CONTROL_EVENT_CANCEL_ALL command, as appropriate. This will indicate whether the cancellation will be with immediate effect or a random delay will be applied:

- If the cancellation is with immediate effect, the application should stop load control for the relevant device(s).
- If a random delay is to be applied to the cancellation, the cluster will put the LCE in the 'Cancelled' list until the delay has expired, when the LCE will be moved to the 'Deallocated' list. Another E_SE_DRLC_EVENT_COMMAND callback event containing the command LOAD_CONTROL_EVENT_CANCEL or LOAD_CONTROL_EVENT_CANCEL_ALL will then be generated, this time indicating immediate cancellation. The application should now stop load control for the relevant device(s).

9.6 Message Signing (Security)

As a security measure, Report Event Status messages can be signed by the DRLC cluster client for non-repudiation purposes (to provide the utility company with evidence that the cluster client sent the message). On the DRLC cluster client, the process involves generating a hash value which is based on the content of the message, then using this value in combination with a device's private key to generate a signature which is then appended to the message to be sent to the ESP.

Upon message reception on the ESP, the hash value is recalculated based on the received message and then used in conjunction with the public key of the message originator (derived from the originator's certificate) to check the appended signature. To facilitate this checking, the ESP must store the certificates of any nodes that send Report Event Status messages which require verification.

Note 1: It is recommended that signatures are supported by applications for backward compatibility.

Note 2: Signature fields are included in the Report Event Status structure, detailed in Section 9.11.4.

Message signing must be enabled at compile-time, as described in Section 9.12.

9.7 DRLC Events

The DRLC cluster has its own events that are handled through the callback mechanism outlined in Section 4.7 (and fully detailed in the *ZCL User Guide (JN-UG-3077)*). If a device uses the DRLC cluster then DRLC event handling must be included in the callback function for the associated endpoint - for example:

- For an ESP (cluster server), this callback function is registered through **eSE_RegisterEspMeterEndPoint()** or **eSE_RegisterEspEndPoint()**
- For an IPD (cluster client), this callback function is registered through **eSE_RegisterIPDEndPoint()**

The relevant callback function will then be invoked when a DRLC event occurs.

For a DRLC event, the `eEventType` field of the `tsZCL_CallBackEvent` structure is set to E_ZCL_CBET_CLUSTER_CUSTOM. This event structure also contains an element `sClusterCustomMessage`, which is itself a structure containing a field `pvCustomData`. This field is a pointer to a `tsSE_DRLCCallBackMessage` structure which contains the DRLC parameters:

```
typedef struct
{
    teSE_DRLCCallBackEventType              eEventType;
    uint8                                   u8CommandId;
    teSE_DRLCStatus                         eDRLCStatus;
    uint32                                  u32CurrentTime;
    union {
        tsSE_DRLCLoadControlEvent           sLoadControlEvent;
        tsSE_DRLCCancelLoadControlEvent     sCancelLoadControlEvent;
        tsSE_DRLCCancelLoadControlAllEvent  sCancelLoadControlAllEvent;
        tsSE_DRLCReportEvent                sReportEvent;
        tsSE_DRLCGetScheduledEvents         sGetScheduledEvents;
    } uMessage;
} tsSE_DRLCCallBackMessage;
```

Information on the elements of the above structure is provided in the sub-sections below.

9.7.1 Event and Command Types

The eEventType field of the tsSE_DRLCCallBackMessage structure above specifies the type of DRLC event that has been generated - these event types are enumerated in the teSE_DRLCCallBackEventType structure, described below.

Note: The u8CommandId field of the tsSE_DRLCCallBackMessage structure is only required for a DRLC event of type E_SE_DRLC_EVENT_COMMAND (see below).

```
typedef enum PACK
{
    E_SE_DRLC_EVENT_API =0x00,
    E_SE_DRLC_EVENT_COMMAND,
    E_SE_DRLC_EVENT_ACTIVE,
    E_SE_DRLC_EVENT_EXPIRED,
    E_SE_DRLC_EVENT_CANCELLED,
    E_SE_DRLC_EVENT_ENUM_END,
} teSE_DRLCCallBackEventType;
```

E_SE_DRLC_EVENT_API

The E_SE_DRLC_EVENT_API event is reserved for internal use.

E_SE_DRLC_EVENT_COMMAND

The E_SE_DRLC_EVENT_COMMAND event is generated when a command has been received on either the server or client. In the tsSE_DRLCCallBackMessage structure, the u8CommandId field is used to indicate the corresponding command - one of:

Command	Description
SE_DRLC_LOAD_CONTROL_EVENT	Generated on a client when a new LCE has been received from the server and added to the 'Scheduled' (or 'Active') list - the LCE is included in the uMessage.LoadControlEvent field of the tsSE_DRLCCallBackMessage structure
SE_DRLC_LOAD_CONTROL_EVENT_CANCEL *	Generated on a client when a command has been received to cancel an LCE and the LCE has been moved to the 'Cancelled' or ' Deallocated' list - which list depends on whether an immediate or randomised end-time is specified in the uMessage.sCancelLoadControlEvent field of the tsSE_DRLCCallBackMessage structure

Table 32: Command Types

Command	Description
SE_DRLC_LOAD_CONTROL_EVENT_CANCEL_ALL *	Generated on a client when a command has been received to cancel all LCEs and the LCEs have been moved to the 'Cancelled' or ' Deallocated' list - which list depends on whether an immediate or randomised end-time is specified in the `uMessage.sCancelLoadControlAllEvent` field of the `tsSE_DRLCCallBackMessage` structure
SE_DRLC_REPORT_EVENT_STATUS **	Generated on the server when a Report Event Status message is received from a client - the contents of the report are included in the `uMessage.sReportEvent` field of the `tsSE_DRLCCallBackMessage` structure
SE_DRLC_GET_SCHEDULED_EVENTS **	Generated on the server when a Get Scheduled Events message is received from a client - the contents of the request are included in the `uMessage.sGetScheduledEvents` field of the `tsSE_DRLCCallBackMessage` structure

Table 32: Command Types

* If an LCE cancellation with a randomised end-time is required, the LCE will first be moved to the 'Cancelled' list and the event will be generated with randomised end-time specified. When the randomised end-time has been reached, the LCE will be moved to the 'Deallocated' list and the event will be generated again but with an immediate end-time specified. The application must then stop the corresponding load control.

** The server can identify which client has sent a Report Event Status or Get Scheduled Events message by examining the `pZPSevent` field of the `tsZCL_CallBackEvent` structure that contains the message.

E_SE_DRLC_EVENT_ACTIVE

The E_SE_DRLC_EVENT_ACTIVE event is generated when an LCE has been moved from the 'Scheduled' list to the 'Active' list (see Section 9.4.2). The activated LCE is included in the `uMessage.LoadControlEvent` field of the `tsSE_DRLCCallBackMessage` structure.

E_SE_DRLC_EVENT_EXPIRED

The E_SE_DRLC_EVENT_EXPIRED event is generated when an LCE has been moved from the 'Active' list (see Section 9.4.2). The expired LCE is included in the `uMessage.LoadControlEvent` field of the `tsSE_DRLCCallBackMessage` structure.

E_SE_DRLC_EVENT_CANCELLED

The E_SE_DRLC_EVENT_CANCELLED event is generated when an LCE has been put in the 'Cancelled' list (see Section 9.4.2) as the result of an LCE 'cancel' or 'cancel all' command. Information on the cancelled LCE(s) is included in the `uMessage.sCancelLoadControlEvent` or `uMessage.sCancelLoadControlAllEvent` field of the `tsSE_DRLCCallBackMessage` structure, as appropriate.

9.7.2 Other Elements of tsSE_DRLCCallBackMessage

In addition to the fields eEventType and u8CommandId described in Section 9.7.1, the tsSE_DRLCCallBackMessage structure contains the following elements.

eDRLCStatus

The eDRLCStatus field indicates the status returned from the command that has been executed (the command identified in u8CommandId). The status codes are enumerated in the teSE_DRLCStatus structure, shown below and described in Section 9.9.

```
typedef enum PACK
{
    E_SE_DRLC_DUPLICATE_EXISTS  = 0x80,
    E_SE_DRLC_EVENT_LATE,
    E_SE_DRLC_EVENT_NOT_YET_ACTIVE,
    E_SE_DRLC_EVENT_OLD,
    E_SE_DRLC_NOT_FOUND,
    E_SE_DRLC_EVENT_NOT_FOUND,
    E_SE_DRLC_EVENT_IGNORED,
    E_SE_DRLC_CANCEL_DEFERRED,
    E_SE_DRLC_BAD_DEVICE_CLASS,
    E_SE_DRLC_BAD_CRITICALITY_LEVEL,
    E_SE_DRLC_DURATION_TOO_LONG,
    E_SE_DRLC_ENUM_END
} teSE_DRLCStatus;
```

u32CurrentTime

The u32CurrentTime field contains the time (UTC) at which the event was generated.

uMessage

This field is a union of structures, containing a structure for each of the DRLC command payloads. The valid structure in the event is defined by the value of u8CommandId (refer to the description of the E_SE_DRLC_EVENT_COMMAND event in Section 9.7.1).

9.8 Functions

The following DRLC cluster functions are provided in the SE API:

Function	Page
eSE_DRLCCreate	251
eSE_DRLCAddLoadControlEvent	253
eSE_DRLCGetScheduledEventsSend	254
eSE_DRLCCancelLoadControlEvent	255
eSE_DRLCCancelAllLoadControlEvents	256
eSE_DRLCSetEventUserOption	257
eSE_DRLCSetEventUserData	258
eSE_DRLCGetLoadControlEvent	259
eSE_DRLCFindLoadControlEvent	260

eSE_DRLCCreate

```
teZCL_Status eSE_DRLCCreate(
                bool_t bIsServer,
                uint8 u8NumberOfRecordEntries,
                uint8 *pu8AttributeControlBits,
                tsZCL_ClusterInstance *psClusterInstance,
                tsZCL_ClusterDefinition *psClusterDefinition,
                tsSE_DRLCCustomDataStructure
                               *psCustomDataStructure,
                tsSE_DRLCLoadControlEventRecord
                               *psDRLCLoadControlEventRecord,
                void *pvEndPointSharedStructPtr);
```

Description

This function creates an instance of the DRLC cluster on an endpoint. The cluster instance is created on the endpoint which is associated with the supplied `tsZCL_ClusterInstance` structure and can act as a server or a client, as specified.

The function should only be called when setting up a custom endpoint containing one or more selected clusters (rather than the whole set of clusters supported by a standard ZigBee device). This function will create a DRLC cluster instance on the endpoint, but instances of other clusters may also be created on the same endpoint by calling their corresponding creation functions. For more details of creating cluster instances on custom endpoints, refer to Appendix B.

Note: This function must not be called for an endpoint on which a standard ZigBee device (e.g. IPD) will be used. In this case, the device and its supported clusters must be registered on the endpoint using the relevant device registration function from those described in Chapter 12.

When used, this function must be the first DRLC cluster function called in the application, and must be called after the stack has been started and after the application profile has been initialised.

The function requires an array to be declared for internal use, which contains one element (of type **uint8**) for each attribute of the cluster. The array length should therefore equate be the total number of attributes supported by the DRLC cluster, which can be obtained by using the macro CLD_DRLC_MAX_NUMBER_OF_ATTRIBUTE.

The array declaration should be as follows:

```
uint8
au8AppDRLC_ClusterAttributeControlBits[CLD_DRLC_MAX_NUMBER_OF_ATTRIBUTE];
```

The function will initialise the array elements to zero.

Parameters

bIsServer	Type of cluster instance (server or client) to be created: TRUE - server FALSE - client
u8NumberOfRecordEntries	Number of LCEs that can be stored in the LCE list, one of: SE_DRLC_NUMBER_OF_SERVER_LOAD_CONTROL_ENTRIES SE_DRLC_NUMBER_OF_CLIENT_LOAD_CONTROL_ENTRIES
pu8AttributeControlBits	Pointer to an array of **uint8** values, with one element for each attribute in the cluster (see above).
psClusterInstance	Pointer to structure containing information about the cluster instance to be created (see the *ZCL User Guide (JN-UG-3077)*). This structure will be updated by the function by initialising individual structure fields.
psClusterDefinition	Pointer to structure indicating the type of cluster to be created (see the *ZCL User Guide (JN-UG-3077)*). In this case, this structure must contain the details of the DRLC cluster. This parameter can refer to a pre-filled structure called `sCLD_DRLC` which is provided in the **DRLC.h** file.
psCustomDataStructure	Pointer to structure which contains custom data for the DRLC cluster. This structure is used for internal data storage. No knowledge of the fields of this structure is required.
psDRLCLoadControlEventRecord	Pointer to a structure in which an LCE will be stored
pvEndPointSharedStructPtr	Pointer to the shared structure used for attribute storage. This parameter should be the address of the structure of type `tsCLD_DRLC` which defines the attributes of DRLC cluster. The function will initialise the attributes with default values.

Returns

E_ZCL_SUCCESS

E_ZCL_FAIL

E_ZCL_ERR_PARAMETER_NULL

E_ZCL_ERR_INVALID_VALUE

eSE_DRLCAddLoadControlEvent

```
teSE_DRLCStatus eSE_DRLCAddLoadControlEvent(
            uint8 u8SourceEndPointId,
            uint8 u8DestinationEndPointId,
            tsZCL_Address psDestinationAddress,
            tsSE_DRLCLoadControlEvent *psLoadControlEvent,
            uint8 *pu8TransactionSequenceNumber);
```

Description

This function can be used on the DRLC cluster server to add an LCE (received from the utility company) to the 'Scheduled' list. The function also sends the LCE to the specified DRLC cluster client endpoints, where it will also be added to the 'Scheduled' list. Note that the LCE will be added to the 'Active' lists on the relevant devices if a 'start-time of now' is specified in the LCE.

The LCE should normally be sent to client endpoints that have been previously bound to the cluster server. This is done by specifying an address type of E_ZCL_AM_BOUND in the `tsZCL_Address` structure - in this case, the address field of this structure and the destination endpoint in the function call are both ignored.

Parameters

u8SourceEndPointId	Number of the local endpoint through which the LCE will be sent
u8DestinationEndPointId	Number of the remote endpoint to which the LCE will be sent. Note that this parameter is ignored when sending to address types E_ZCL_AM_BOUND and E_ZCL_AM_GROUP
psDestinationAddress	Pointer to a ZCL structure containing the address of the remote node to which the LCE will be sent
psLoadControlEvent	Pointer to a structure (see Section 9.11.1) which contains the LCE to be added and sent
pu8TransactionSequenceNumber	Pointer to a location to store the Transaction Sequence Number (TSN) of the packet sent

Returns

Any relevant DRLC return code listed in Section 9.9 or ZCL return code listed in the *ZCL User Guide (JN-UG-3077)*

eSE_DRLCGetScheduledEventsSend

teSE_DRLCStatus eSE_DRLCGetScheduledEventsSend(
uint8 *u8SourceEndPointId*,
uint8 *u8DestinationEndPointId*,
tsZCL_Address *psDestinationAddress*,
tsSE_DRLCGetScheduledEvents **psGetScheduledEvents*,
uint8 **pu8TransactionSequenceNumber***);**

Description

This function can be used on a DRLC cluster client to send a Get Scheduled Events message to the cluster server in order to request a list of scheduled (and active) LCEs. The function can be used to obtain the initial schedule of LCEs and to update the local LCE lists during operation (for example, if an End Device has been sleeping and has missed unsolicited LCE updates) - refer to Section 9.5.2.2 for more information on the use of this function.

As part of this function call, a `tsSE_DRLCGetScheduledEvents` structure must be provided which specifies the earliest start-time of the LCEs of interest and the maximum number of LCEs to be reported.

Parameters

u8SourceEndPointId	Number of the local endpoint through which the request will be sent
u8DestinationEndPointId	Number of the remote endpoint to which the request will be sent (this must be the DRLC cluster server endpoint)
psDestinationAddress	Pointer to a ZCL structure containing the address of the remote node to which the request will be sent (this must be the address of the ESP)
psGetScheduledEvents	Pointer to a structure which contains the LCE requirements of the request (see Section 9.11.2)
pu8TransactionSequenceNumber	Pointer to a location to store the Transaction Sequence Number (TSN) of the request

Returns

Any relevant DRLC return code listed in Section 9.9 or ZCL return code listed in the *ZCL User Guide (JN-UG-3077)*

eSE_DRLCCancelLoadControlEvent

```
teSE_DRLCStatus eSE_DRLCCancelLoadControlEvent(
        uint8 u8SourceEndPointId,
        uint8 u8DestinationEndPointId,
        tsZCL_Address psDestinationAddress,
        tsSE_DRLCCancelLoadControlEvent
                                *psCancelLoadControlEvent,
        uint8 *pu8TransactionSequenceNumber);
```

Description

This function can be used on the DRLC cluster server to cancel an LCE. The LCE will be cancelled locally and the cancellation will also be sent to the specified DRLC cluster client endpoints. The LCE will be ultimately moved to the 'Deallocated' list.

The cancellation request should normally be sent to client endpoints that have been previously bound to the cluster server. This is done by specifying an address type of E_ZCL_AM_BOUND in the `tsZCL_Address` structure - in this case, the address field of this structure and the destination endpoint in the function call are both ignored.

The LCE cancellation requirements are specified in the structure `tsSE_DRLCCancelLoadControlEvent`, including the applicable device class(es) and enrolment group(s), as well as an immediate or randomised end (for a full description of the end-time options, refer to Section 9.5.3).

Parameters

u8SourceEndPointId	Number of the local endpoint through which the request will be sent
u8DestinationEndPointId	Number of the remote endpoint to which the request will be sent. Note that this parameter is ignored when sending to address types E_ZCL_AM_BOUND and E_ZCL_AM_GROUP
psDestinationAddress	Pointer to a ZCL structure containing the address of the remote node to which the request will be sent
psCancelLoadControlEvent	Pointer to a structure which contains the LCE cancellation requirements (see Section 9.11.3)
pu8TransactionSequenceNumber	Pointer to a location to store the Transaction Sequence Number (TSN) of the request

Returns

Any relevant DRLC return code listed in Section 9.9 or ZCL return code listed in the *ZCL User Guide (JN-UG-3077)*

eSE_DRLCCancelAllLoadControlEvents

teSE_DRLCStatus eSE_DRLCCancelAllLoadControlEvents(
uint8 *u8SourceEndPointId*,
uint8 *u8DestinationEndPointId*,
tsZCL_Address *psDestinationAddress*,
teSE_DRLCCancelControl *eCancelEventControl*,
uint8 ****pu8TransactionSequenceNumber*);**

Description

This function can be used on the DRLC cluster server to cancel all LCEs. The LCEs will be cancelled locally and the cancellation will also be sent to the specified DRLC cluster client endpoints. The LCEs will be ultimately moved to the ‘Deallocated’ list.

The cancellation request should normally be sent to client endpoints that have been previously bound to the cluster server. This is done by specifying an address type of E_ZCL_AM_BOUND in the `tsZCL_Address` structure - in this case, the address field of this structure and the destination endpoint in the function call are both ignored.

The LCE cancellation end-time requirement must be specified as an immediate or randomised end (for a full description of the end-time options, refer to Section 9.5.3).

Parameters

u8SourceEndPointId	Number of the local endpoint through which the request will be sent
u8DestinationEndPointId	Number of the remote endpoint to which the request will be sent. Note that this parameter is ignored when sending to address types E_ZCL_AM_BOUND and E_ZCL_AM_GROUP
psDestinationAddress	Pointer to a ZCL structure containing the address of the remote node to which the request will be sent
eCancelEventControl	Enumeration indicating an immediate or randomised end, one of: E_SE_DRLC_CANCEL_CONTROL_IMMEDIATE E_SE_DRLC_CANCEL_CONTROL_USE_RANDOMISATION
pu8TransactionSequenceNumber	Pointer to a location to store the Transaction Sequence Number (TSN) of the request

Returns

Any relevant DRLC return code listed in Section 9.9 or ZCL return code listed in the *ZCL User Guide (JN-UG-3077)*

eSE_DRLCSetEventUserOption

```
teSE_DRLCStatus eSE_DRLCSetEventUserOption(
            uint32 u32IssuerId,
            uint8 u8SourceEndPointId,
            teSE_DRLCUserEventOption eEventOption);
```

Description

This function can be used on a DRLC cluster client to choose to participate or not participate in an individual LCE. By default, a client participates in an LCE, so normally this function only needs to be called if the client is to opt out of the LCE.

The function could be called following a button-press which results from a user decision to opt out of the LCE (for which information is displayed on the IPD screen).

When this function is called, a Report Event Status message is sent to the cluster server in order to indicate that the local client has opted out of the LCE. Once the LCE end-time has been reached, another Report Event Status message is sent to the server in order to confirm that the LCE has completed without the participation of the local client.

Parameters

u32IssuerId	Identifier of the LCE (as issued by the utility company)
u8SourceEndPointId	Number of the local endpoint where the LCE is located (endpoint corresponding to the DRLC cluster)
eEventOption	Required option, one of: E_SE_DRLC_EVENT_USER_OPT_IN (participate) E_SE_DRLC_EVENT_USER_OPT_OUT (do not participate)

Returns

Any relevant DRLC return code listed in Section 9.9 or ZCL return code listed in the *ZCL User Guide (JN-UG-3077)*

eSE_DRLCSetEventUserData

```
teSE_DRLCStatus eSE_DRLCSetEventUserData(
                    uint32 u32IssuerId,
                    uint8 u8SourceEndPointId,
                    teSE_DRLCUserEventSet eUserEventSetID,
                    uint16 u16EventData);
```

Description

This function can be used on a DRLC cluster client to locally modify the load control data of an LCE. Any one of the following data values can be changed:

- Criticality level
- Cooling temperature set-point
- Heating temperature set-point
- Load adjustment percentage
- Duty cycle

The function can be called multiple times to modify more than one of the above values. The data values are fully described in Section 9.11.1.

Parameters

u32IssuerId	Identifier of the LCE (as issued by the utility company)
u8SourceEndPointId	Number of the local endpoint where the LCE is located (endpoint corresponding to the DRLC cluster)
eUserEventSetID	Identifier of the load control data item to be modified, one of: E_SE_DRLC_CRITICALITY_LEVEL_APPLIED E_SE_DRLC_COOLING_TEMPERATURE_SET_POINT_APPLIED E_SE_DRLC_HEATING_TEMPERATURE_SET_POINT_APPLIED E_SE_DRLC_AVERAGE_LOAD_ADJUSTMENT_PERCENTAGE_APPLIED E_SE_DRLC_DUTY_CYCLE_APPLIED
u16EventData	Value to which the specified data item is to be set (for formats of data values, refer to descriptions in Section 9.11.1)

Returns

Any relevant DRLC return code listed in Section 9.9 or ZCL return code listed in the *ZCL User Guide (JN-UG-3077)*

eSE_DRLCGetLoadControlEvent

```
teSE_DRLCStatus eDRLCGetLoadControlEvent(
        uint8 u8SourceEndPointId,
        uint8 u8TableIndex,
        teSE_DRLCEventList eEventList,
        tsSE_DRLCLoadControlEvent **ppsLoadControlEvent);
```

Description

This function can be used to obtain an LCE from a local LCE list.

The required list must be specified as one of ‘Scheduled’, ‘Active’, ‘Cancelled’ and ‘Deallocated’. The index of the required LCE in the list must also be specified. The index of zero is used to indicate that the LCE with the oldest start-time should be retrieved. To retrieve all the LCEs in a list, repeatedly call this function with index zero until the function indicates that there are no further LCEs in the list (returns E_SE_DRLC_EVENT_NOT_FOUND).

Parameters

u8SourceEndPointId	Number of the local endpoint from which the LCE is to be retrieved (endpoint corresponding to the DRLC cluster)
u8TableIndex	Index of required LCE in the specified LCE list (see below)
eEventList	LCE list from which the LCE is to be retrieved, one of: E_SE_DRLC_EVENT_LIST_SCHEDULED E_SE_DRLC_EVENT_LIST_ACTIVE E_SE_DRLC_EVENT_LIST_CANCELLED E_SE_DRLC_EVENT_LIST_DEALLOCATED
ppsLoadControlEvent	Pointer to a pointer to a `tsSE_DRLCLoadControlEvent` structure to receive the obtained LCE (see Section 9.11.1)

Returns

Any relevant DRLC return code listed in Section 9.9 or ZCL return code listed in the *ZCL User Guide (JN-UG-3077)*

eSE_DRLCFindLoadControlEvent

```
teSE_DRLCStatus  eSE_DRLCFindLoadControlEvent(
        uint8 u8SourceEndPointId,
        uint32 u32IssuerId,
        bool_t bIsServer,
        tsSE_DRLCLoadControlEvent **ppsLoadControlEvent,
        teSE_DRLCEventList *peEventList);
```

Description

This function can be used to obtain the specified LCE from the local LCE lists.

The required LCE must be specified in terms of its identifier issued by the utility company. The function will search all the local LCE lists, identify the list (if any) in which the LCE was found and return the found LCE.

Parameters

u8SourceEndPointId	Number of the local endpoint from which the LCE is to be retrieved (endpoint corresponding to the DRLC cluster)
u32IssuerId	Identifier of the LCE to be found (as issued by the utility company)
bIsServer	Cluster server or client: TRUE - server FALSE - client
ppsLoadControlEvent	Pointer to a pointer to a `tsSE_DRLCLoadControlEvent` structure to receive the obtained LCE (see Section 9.11.1)
peEventList	Pointer to variable to receive enumerated value of the list in which the LCE was found (see Section 9.10.7)

Returns

Any relevant DRLC return code listed in Section 9.9 or ZCL return code listed in the *ZCL User Guide (JN-UG-3077)*

9.9 Return Codes

In addition to some of the ZCL status enumerations (detailed in the *ZCL User Guide (JN-UG-3077)*), the following enumerations are returned by the DRLC cluster functions (described in Section 9.8) to indicate the outcome of the function call.

```
typedef enum PACK
{
    E_SE_DRLC_DUPLICATE_EXISTS = 0x80,
    E_SE_DRLC_EVENT_LATE,
    E_SE_DRLC_EVENT_NOT_YET_ACTIVE,
    E_SE_DRLC_EVENT_OLD,
    E_SE_DRLC_NOT_FOUND,
    E_SE_DRLC_EVENT_NOT_FOUND,
    E_SE_DRLC_EVENT_IGNORED,
    E_SE_DRLC_CANCEL_DEFERRED,
    E_SE_DRLC_BAD_DEVICE_CLASS,
    E_SE_DRLC_BAD_CRITICALITY_LEVEL,
    E_SE_DRLC_DURATION_TOO_LONG,
    E_SE_DRLC_ENUM_END
} teSE_DRLCStatus;
```

The above return codes are described in the table below.

Enumeration	Description
E_SE_DRLC_DUPLICATE_EXISTS	An overlapping LCE (in time) has been found
E_SE_DRLC_EVENT_LATE	Function call refers to a time period that is earlier than the current ZCL time
E_SE_DRLC_EVENT_NOT_YET_ACTIVE	Not used - reserved for future use
E_SE_DRLC_EVENT_OLD	Not used - reserved for future use
E_SE_DRLC_NOT_FOUND	LCE cannot be found in lists (used in LCE cancellation or opt out)
E_SE_DRLC_EVENT_NOT_FOUND	LCE cannot be found in lists (used when searching for an LCE)
E_SE_DRLC_EVENT_IGNORED	Not used - reserved for future use
E_SE_DRLC_CANCEL_DEFERRED	Cancellation has been processed but is deferred to act in the future
E_SE_DRLC_BAD_DEVICE_CLASS	Specified device class not recognised
E_SE_DRLC_BAD_CRITICALITY_LEVEL	Specified criticaility level not recognised
E_SE_DRLC_DURATION_TOO_LONG	Specified duration exceeds maximum of 1440 minutes (one day)

Table 33: Return Codes

9.10 Enumerations

9.10.1 'Device Class' Enumerations

The device classes that are used in load control are enumerated in the `teSE_DRLCDeviceClassFieldBitmap` structure below:

```
typedef enum
{
    E_SE_DRLC_HVAC_COMPRESSOR_OR_FURNACE_BIT = 0x00,
    E_SE_DRLC_STRIP_BASEBOARD_HEATERS_BIT,
    E_SE_DRLC_WATER_HEATER_BIT,
    E_SE_DRLC_POOL_PUMP_SPA_JACUZZI_BIT,
    E_SE_DRLC_SMART_APPLIANCES_BIT,
    E_SE_DRLC_IRRIGATION_PUMP_BIT,
    E_SE_DRLC_MANAGED_COMMERCIAL_AND_INDUSTRIAL_LOADS_BIT,
    E_SE_DRLC_SIMPLE_MISC_LOADS_BIT,
    E_SE_DRLC_EXTERIOR_LIGHTING_BIT,
    E_SE_DRLC_INTERIOR_LIGHTING_BIT,
    E_SE_DRLC_ELECTRIC_VEHICLE_BIT,
    E_SE_DRLC_GENERATION_SYSTEMS_BIT,
    E_SE_DRLC_DEVICE_CLASS_FIRST_RESERVED_BIT
} teSE_DRLCDeviceClassFieldBitmap;
```

The device class enumerations are listed and described in the table below.

Device Class Enumeration	**Description**
E_SE_DRLC_HVAC_COMPRESSOR_OR_FURNACE_BIT	HVAC compressor or furnace
E_SE_DRLC_STRIP_BASEBOARD_HEATERS_BIT	Strip/baseboard heater
E_SE_DRLC_WATER_HEATER_BIT	Water heater
E_SE_DRLC_POOL_PUMP_SPA_JACUZZI_BIT	Pool/spa/jacuzzi pump
E_SE_DRLC_SMART_APPLIANCES_BIT	Smart appliance
E_SE_DRLC_IRRIGATION_PUMP_BIT	Irrigation pump
E_SE_DRLC_MANAGED_COMMERCIAL_AND_INDUSTRIAL_ LOADS_BIT	Managed Commercial & Industrial (C&I)
E_SE_DRLC_SIMPLE_MISC_LOADS_BIT	Simple miscellaneous (residential on/off)
E_SE_DRLC_EXTERIOR_LIGHTING_BIT	Exterior lighting
E_SE_DRLC_INTERIOR_LIGHTING_BIT	Interior lighting
E_SE_DRLC_ELECTRIC_VEHICLE_BIT	Electric vehicle

Table 34: Device Classes

Device Class Enumeration	Description
E_SE_DRLC_GENERATION_SYSTEMS_BIT	Generation systems
E_SE_DRLC_DEVICE_CLASS_FIRST_RESERVED_BIT	Reserved

Table 34: Device Classes

9.10.2 ‘DRLC Event’ Enumerations

The event types generated by the DRLC cluster are enumerated in the `teSE_DRLCCallBackEventType` structure below:

```
typedef enum PACK
{
    E_SE_DRLC_EVENT_API =0x00,
    E_SE_DRLC_EVENT_COMMAND,
    E_SE_DRLC_EVENT_ACTIVE,
    E_SE_DRLC_EVENT_EXPIRED,
    E_SE_DRLC_EVENT_CANCELLED,
    E_SE_DRLC_EVENT_ENUM_END,
} teSE_DRLCCallBackEventType;
```

The above event types are described in the table below.

Event Type Enumeration	Description
E_SE_DRLC_EVENT_API	Reserved for internal use
E_SE_DRLC_EVENT_COMMAND	Generated when a command has been received from either the cluster server or a cluster client
E_SE_DRLC_EVENT_ACTIVE	Generated when an LCE has been added to the ‘Active’ list
E_SE_DRLC_EVENT_EXPIRED	Generated when an LCE has been removed from the ‘Active’ list
E_SE_DRLC_EVENT_CANCELLED	Generated when an LCE has been put in the ‘Cancelled’ list

Table 35: DRLC Event Types

DRLC events are described in more detail in Section 9.7.

9.10.3 'Criticality Level' Enumerations

The criticality levels that are available for an LCE are enumerated in the `teSE_DRLCCriticalityLevels` structure below:

```
typedef enum
{
    E_SE_DRLC_RESERVED_0_CRITICALITY = 0x00,
    E_SE_DRLC_GREEN_CRITICALITY,
    E_SE_DRLC_VOLUNTARY_1_CRITICALITY,
    E_SE_DRLC_VOLUNTARY_2_CRITICALITY,
    E_SE_DRLC_VOLUNTARY_3_CRITICALITY,
    E_SE_DRLC_VOLUNTARY_4_CRITICALITY,
    E_SE_DRLC_VOLUNTARY_5_CRITICALITY,
    E_SE_DRLC_EMERGENCY_CRITICALITY,
    E_SE_DRLC_PLANNED_OUTAGE_CRITICALITY,
    E_SE_DRLC_SERVICE_DISCONNECT_CRITICALITY,
    E_SE_DRLC_UTILITY_DEFINED_1_CRITICALITY,
    E_SE_DRLC_UTILITY_DEFINED_2_CRITICALITY,
    E_SE_DRLC_UTILITY_DEFINED_3_CRITICALITY,
    E_SE_DRLC_UTILITY_DEFINED_4_CRITICALITY,
    E_SE_DRLC_UTILITY_DEFINED_5_CRITICALITY,
    E_SE_DRLC_UTILITY_DEFINED_6_CRITICALITY,
    E_SE_DRLC_FIRST_RESERVED_CRITICALITY
} teSE_DRLCCriticalityLevels;
```

The above criticality levels are described in the table below.

Criticality Level Enumeration	Description
E_SE_DRLC_RESERVED_0_CRITICALITY	Reserved for future use
E_SE_DRLC_GREEN_CRITICALITY	**Green:** Indicates that there will be a significant contribution from non-green sources during the LCE - participation in the LCE is voluntary
E_SE_DRLC_VOLUNTARY_1_CRITICALITY	**Voluntary 1-6:** Represent increasing levels of load reduction as move through levels 1 to 6, as defined by the utility company - intended to be used in a sequence of LCEs to gradually reduce loads, where participation in the LCEs is voluntary
E_SE_DRLC_VOLUNTARY_2_CRITICALITY	
E_SE_DRLC_VOLUNTARY_3_CRITICALITY	
E_SE_DRLC_VOLUNTARY_4_CRITICALITY	
E_SE_DRLC_VOLUNTARY_5_CRITICALITY	
E_SE_DRLC_EMERGENCY_CRITICALITY	**Emergency:** Indicates that the LCE represents an emergency situation (normally demanding the termination of all non-essential loads, as defined by the utility company) - participation in the LCE is mandatory

Table 36: Criticality Levels

Criticality Level Enumeration	Description
E_SE_DRLC_PLANNED_OUTAGE_CRITICALITY	**Planned Outage:** Indicates that the LCE represents an intentional outage (normally demanding the termination of all non-essential loads, as defined by the utility company) - participation in the LCE is mandatory
E_SE_DRLC_SERVICE_DISCONNECT_CRITICALITY	**Service Disconnect:** Indicates that the LCE represents a service disconnection (normally demanding the termination of all non-essential loads, as defined by the utility company) - participation in the LCE is mandatory
E_SE_DRLC_UTILITY_DEFINED_1_CRITICALITY	**Utility-defined 1-6:** Criticality levels completely defined by the utility company - participation in the LCE is voluntary
E_SE_DRLC_UTILITY_DEFINED_2_CRITICALITY	
E_SE_DRLC_UTILITY_DEFINED_3_CRITICALITY	
E_SE_DRLC_UTILITY_DEFINED_4_CRITICALITY	
E_SE_DRLC_UTILITY_DEFINED_5_CRITICALITY	
E_SE_DRLC_UTILITY_DEFINED_6_CRITICALITY	
E_SE_DRLC_FIRST_RESERVED_CRITICALITY	Reserved for future use

Table 36: Criticality Levels

9.10.4 ‘LCE Cancellation’ Enumerations

The cancellation options (immediate or randomised) that are available for an LCE are enumerated in the `teSE_DRLCCancelControl` structure below:

```
typedef enum PACK
{
    E_SE_DRLC_CANCEL_CONTROL_IMMEDIATE =0x00,
    E_SE_DRLC_CANCEL_CONTROL_USE_RANDOMISATION =0x10
} teSE_DRLCCancelControl;
```

The above options are described in the table below.

LCE Cancellation Enumeration	Description
E_SE_DRLC_CANCEL_CONTROL_IMMEDIATE	LCE will be cancelled immediately by moving it directly to the ‘Deallocated’ list - a randomised end-time configured in the LCE will be ignored
E_SE_DRLC_CANCEL_CONTROL_USE_RANDOMISATION	A random delay will be applied to the LCE cancellation, if a randomised end-time was configured in the LCE - the LCE will be moved to the ‘Cancelled’ list where it will stay (and remain valid) until the random delay has expired, when it will be moved to the ‘Deallocated’ list (an upper limit on the delay is defined in the cluster - see Section 9.2)

Table 37: LCE Cancellation Options

9.10.5 'LCE Participation' Enumerations

The options to participate or not participate in an LCE are enumerated in the `teSE_DRLCUserEventOption` structure below:

```
typedef enum PACK
{
    E_SE_DRLC_EVENT_USER_OPT_IN =0x00,
    E_SE_DRLC_EVENT_USER_OPT_OUT
} teSE_DRLCUserEventOption;
```

The above options are described in the table below.

LCE Participation Enumeration	Description
E_SE_DRLC_EVENT_USER_OPT_OUT	User has opted not to participate in the LCE. The device sends this message and does not adjust the load when the LCE becomes active.
E_SE_DRLC_EVENT_USER_OPT_IN	User has opted to participate in the LCE. The device only sends this message following an OPT_OUT (when the user has changed their mind and decided to participate after all)

Table 38: LCE Participation Options

9.10.6 'LCE Data Modification' Enumerations

The load control data items that can be locally modified in an LCE are enumerated in the `teSE_DRLCUserEventSet` structure below:

```
typedef enum PACK
{
    E_SE_DRLC_CRITICALITY_LEVEL_APPLIED =0x00,
    E_SE_DRLC_COOLING_TEMPERATURE_SET_POINT_APPLIED,
    E_SE_DRLC_HEATING_TEMPERATURE_SET_POINT_APPLIED,
    E_SE_DRLC_AVERAGE_LOAD_ADJUSTMENT_PERCENTAGE_APPLIED,
    E_SE_DRLC_DUTY_CYCLE_APPLIED,
    E_SE_DRLC_USER_EVENT_ENUM_END,
} teSE_DRLCUserEventSet;
```

The above options are described in the table below (the data items are fully described in Section 9.11.1).

LCE Participation Enumeration	Description
E_SE_DRLC_CRITICALITY_LEVEL_APPLIED	Specifies that 'criticality level' is to be modified.
E_SE_DRLC_COOLING_TEMPERATURE_ SET_POINT_APPLIED	Specifies that 'cooling temperature set-point' is to be modified
E_SE_DRLC_HEATING_TEMPERATURE_ SET_POINT_APPLIED	Specifies that 'heating temperature set-point' is to be modified
E_SE_DRLC_AVERAGE_LOAD_ADJUSTMENT_ PERCENTAGE_APPLIED	Specifies that 'average load adjustment percentage' is to be modified
E_SE_DRLC_DUTY_CYCLE_APPLIED	Specifies that 'duty cycle' is to be modified

Table 39: LCE Data Modification Options

9.10.7 'LCE List' Enumerations

The LCE lists are enumerated in the `teSE_DRLCEventList` structure below:

```
typedef enum PACK
{
    E_SE_DRLC_EVENT_LIST_SCHEDULED =0x00,
    E_SE_DRLC_EVENT_LIST_ACTIVE,
    E_SE_DRLC_EVENT_LIST_CANCELLED,
    E_SE_DRLC_EVENT_LIST_DEALLOCATED,
    E_SE_DRLC_EVENT_LIST_NONE
} teSE_DRLCEventList;
```

The above lists are described in the table below.

LCE List Enumeration	Description
E_SE_DRLC_EVENT_LIST_SCHEDULED	**Scheduled list:** Contains LCEs that are due to be executed in the future
E_SE_DRLC_EVENT_LIST_ACTIVE	**Active list:** Contains LCEs that are currently being executed
E_SE_DRLC_EVENT_LIST_CANCELLED	**Cancelled list:** Contains LCEs that have been cancelled with a randomised end-time and whose random end-time has not yet been reached
E_SE_DRLC_EVENT_LIST_DEALLOCATED	**Deallocated list:** Contains expired LCEs and therefore a record of the free storage for LCEs

Table 40: LCE Lists

9.10.8 'LCE Status' Enumerations

LCE status is enumerated in the `teSE_DRLCEventStatus` structure below:

```
typedef enum PACK
{
    E_SE_DRLC_LOAD_CONTROL_EVENT_COMMAND_RECIEVED =0x01,
    E_SE_DRLC_EVENT_STARTED,
    E_SE_DRLC_EVENT_COMPLETED,
    E_SE_DRLC_USER_CHOSEN_OPT_OUT,
    E_SE_DRLC_USER_CHOSEN_OPT_IN,
    E_SE_DRLC_EVENT_HAS_BEEN_CANCELLED,
    E_SE_DRLC_EVENT_HAS_BEEN_SUPERSEDED,
    E_SE_DRLC_EVENT_PARTIALLY_COMPLETED_WITH_USER_OPT_OUT,
    E_SE_DRLC_EVENT_PARTIALLY_COMPLETED_WITH_USER_OPT_IN,
    E_SE_DRLC_EVENT_COMPLETED_NO_USER_PARTICIPATION,
    E_SE_DRLC_REJECTED_INVALID_CANCEL_COMMAND_DEFAULT =0xF8,
    E_SE_DRLC_REJECTED_INVALID_CANCEL_COMMAND_INVALID_EFFECTIVE_TIME,
    E_SE_DRLC_REJECTED_EVENT_RECEIVED_AFTER_IT_HAD_EXPIRED =0xFB,
    E_SE_DRLC_REJECTED_INVALID_CANCEL_COMMAND_UNDEFINED_EVENT=0xFD,
    E_SE_DRLC_LOAD_CONTROL_EVENT_COMMAND_REJECTED,
    E_SE_DRLC_EVENT_STATUS_ENUM_END
} teSE_DRLCEventStatus;
```

The above enumerations are described in the table below.

Enumeration	Description
E_SE_DRLC_LOAD_CONTROL_EVENT_COMMAND_RECEIVED	LCE command received (to add new LCE to local lists)
E_SE_DRLC_EVENT_STARTED	LCE has started
E_SE_DRLC_EVENT_COMPLETED	LCE has completed
E_SE_DRLC_USER_CHOSEN_OPT_OUT	Client has opted out of the LCE
E_SE_DRLC_USER_CHOSEN_OPT_IN	Client has opted into the LCE
E_SE_DRLC_EVENT_HAS_BEEN_CANCELLED	LCE has been cancelled
E_SE_DRLC_EVENT_HAS_BEEN_SUPERSEDED	LCE has been replaced with another LCE
E_SE_DRLC_EVENT_PARTIALLY_COMPLETED_WITH_USER_OPT_OUT	LCE has prematurely completed due to a client opt-out during the LCE
E_SE_DRLC_EVENT_PARTIALLY_COMPLETED_WITH_USER_OPT_IN	LCE has completed but was only partially executed due to a client opt-in during the LCE
E_SE_DRLC_EVENT_COMPLETED_NO_USER_PARTICIPATION	LCE has completed but there was no client participation (due to a client opt-out from the start)
E_SE_DRLC_REJECTED_INVALID_CANCEL_COMMAND_DEFAULT	Received 'cancel command' invalid and rejected (default)

Table 41: LCE Status Codes

Enumeration	Description
E_SE_DRLC_REJECTED_INVALID_CANCEL_ COMMAND_INVALID_EFFECTIVE_TIME	Received ‘cancel command’ rejected due to invalid effective time (start-time of cancellation)
E_SE_DRLC_REJECTED_EVENT_RECEIVED_ AFTER_IT_HAD_EXPIRED	LCE was received after it had expired (current time is greater than start-time + duration)
E_SE_DRLC_REJECTED_INVALID_CANCEL_ COMMAND_UNDEFINED_EVENT	Received ‘cancel command’ due to undefined LCE
E_SE_DRLC_LOAD_CONTROL_EVENT_COMMAND_ REJECTED	LCE command rejected

Table 41: LCE Status Codes

9.11 Structures

9.11.1 tsSE_DRLCLoadControlEvent

The structure of type `tsSE_DRLCLoadControlEvent` contains the parameters of a Load Control Event (LCE), as shown and described below.

```
typedef struct {
    uint8               u8UtilityEnrolmentGroup;
    uint8               u8CriticalityLevel;
    uint8               u8CoolingTemperatureOffset;
    uint8               u8HeatingTemperatureOffset;
    uint8               u8AverageLoadAdjustmentSetPoint;
    uint8               u8DutyCycle;
    uint8               u8EventControl;
    uint16              u16DeviceClass;
    uint16              u16DurationInMinutes;
    uint16              u16CoolingTemperatureSetPoint;
    uint16              u16HeatingTemperatureSetPoint;
    uint32              u32IssuerId;
    uint32              u32StartTime;
} tsSE_DRLCLoadControlEvent;
```

where:

- `u8UtilityEnrolmentGroup` identifies the group of devices to which the LCE applies - an 'enrolment group' is defined by the utility company. The identifier 0x00 is reserved to indicate 'all groups'.
- `u8CriticalityLevel` is a value representing the level of criticality of the LCE. Enumerations are provided for the different levels and are detailed in Section 9.10.3.
- `u8CoolingTemperatureOffset` (optional) specifies the required temperature offset, in units of 0.1^{o}C, above the current temperature set-point of a cooling device (e.g. 0x5 represents a temperature offset of 0.5^{o}C). The setting 0xFF is used to indicate that no offset is required.
- `u8HeatingTemperatureOffset` (optional) specifies the required temperature offset, in units of 0.1^{o}C, below the current temperature set-point of a heating device (e.g. 0x14 represents a temperature offset of 2.0^{o}C). The setting 0xFF is used to indicate that no offset is required.
- `u8AverageLoadAdjustmentSetPoint` (optional) specifies the maximum permissible load as an offset from the consumer's average load, where this offset is expressed as a positive or negative percentage in units of 1% (e.g. 20% allows loads of up to 120% of the average while -10% allows loads of up to 90% of the average). The offset has a valid range of -100% to +100% and is represented in two's complement form (e.g. 15% is represented by 0x0F and

-5% is represented by 0xFB). The value 0x80 is used to indicate that no such limit is required.

- `u8DutyCycle` (optional) specifies the percentage duty cycle for the load supplied to the device - that is, the percentage of the LCE duration for which the load will be supplied (e.g. for a duty cycle of 80%, the supplied device will be 'on' for 80% of the duration of the LCE). The manner in which the duty cycle is implemented (e.g. periodicity) is device-specific. The valid range of duty cycle values is 0 to 100. The setting 0xFF indicates that no duty cycling is required.
- `u8EventControl` specifies whether a randomised start-time and/or randomised end-time are required for the LCE. The following bit-masks are provided to allow the start-time and end-time of an LCE to be individually randomised (they can be bitwise-ORed to randomise both times):

```
#define SE_DRLC_CONTROL_RANDOMISATION_START_TIME_MASK   (0x01)
#define SE_DRLC_CONTROL_RANDOMISATION_STOP_TIME_MASK    (0x02)
```

- `u16DeviceClass` identifies the class(es) of device to which the LCE applies. Enumerations are provided for the various device classes, which may be combined in a bitwise-OR operation, and are detailed in Section 9.10.1.
- `u16DurationInMinutes` specifies the duration, in minutes, of the LCE (although the actual duration will be longer than the specified duration if a randomised end-time is required). The maximum possible duration that can be specified is 1440 minutes (one day).
- `u16CoolingTemperatureSetPoint` (optional) specifies the required temperature set-point, in units of 0.01^{o}C, for a cooling device, where a negative temperature is represented in two's complement form (e.g. a temperature of 20^{o}C is represented by 0x07D0 and -40^{o}C is represented by 0xF060). The valid temperature range is -273.15°C to 327.67°C. The setting 0x8000 is used to indicate that no temperature set-point is required.
- `u16HeatingTemperatureSetPoint` (optional) specifies the required temperature set-point, in units of 0.01^{o}C, for a heating device, where a negative temperature is represented in two's complement form (e.g. a temperature of 25^{o}C is represented by 0x09C4 and -1^{o}C is represented by 0xFFFF). The valid temperature range is -273.15°C to 327.67°C. The setting 0x8000 is used to indicate that no temperature set-point is required.
- `u32IssuerId` is a unique identifier for the LCE, issued by the utility company (the value could be based on the time-stamp of when the LCE was issued).
- `u32StartTime` represents the start-time (UTC) of the LCE (although the actual start-time will be later if a randomised start-time is required). The value 0x000000000 is used to indicate a 'start-time of now'.

9.11.2 tsSE_DRLCGetScheduledEvents

The structure of type `tsSE_DRLCGetScheduledEvents` contains the parameters of a **Get Scheduled Event** message, as shown and described below.

```
typedef struct {
    uint32              u32StartTime;
    uint8               u8numberOfEvents;
} tsSE_DRLCGetScheduledEvents;
```

where:

- `u32StartTime` is the earliest start-time (UTC) of the requested LCEs
- `u8numberOfEvents` is the maximum number of LCEs to report

9.11.3 tsSE_DRLCCancelLoadControlEvent

The structure of type `tsSE_DRLCCancelLoadControlEvent` contains the parameters of a Cancel LCE command, as shown and described below.

```
typedef struct {
    uint32                  u32IssuerId;
    uint16                  u16DeviceClass;
    uint8                   u8UtilityEnrolmentGroup;
    teSE_DRLCCancelControl  eCancelControl;
    uint32                  u32effectiveTime;
} tsSE_DRLCCancelLoadControlEvent;
```

where:

- `u32IssuerId` is the identifer (provided by the utility company) of the LCE to be cancelled
- `u16DeviceClass` is a bitmap indicating the device class(es) to which the LCE cancellation applies - enumerations for the device classes are provided, as described in Section 9.10.1
- `u8UtilityEnrolmentGroup` is the enrolment group of the devices to which the LCE cancellation applies
- `eCancelControl` indicates whether to honour a randomised end that has been configured in the LCE - enumerations are provided, as described in Section 9.10.4
- `u32effectiveTime` is the time (UTC) from which the LCE cancellation will be effective

9.11.4 tsSE_DRLCReportEvent

The structure of type `tsSE_DRLCReportEvent` contains the parameters of a Report Event Status message, as shown and described below.

```
typedef struct {
    uint8               u8EventStatus;
    uint8               u8AverageLoadAdjustmentPercentageApplied;
    uint8               u8DutyCycleApplied;
    uint8               u8EventControl;
    uint8               u8SignatureType;
    uint8               u8CriticalityLevelApplied;
    bool_t              bSignatureVerified;
    uint16              u16CoolingTemperatureSetPointApplied;
    uint16              u16HeatingTemperatureSetPointApplied;
    uint32              u32IssuerId;
    uint32              u32EventStatusTime;
    tsSE_DRLCOctets     sSignature;
} tsSE_DRLCReportEvent;
```

where:

- `u8EventStatus` is the reported LCE status - enumerations are provided and described in Section 9.10.8
- `u8AverageLoadAdjustmentPercentageApplied` is an optional field containing the load adjustment percentage applied by the sending client (if the user has chosen to over-ride the original setting in the LCE) - for the format of this setting, refer to the equivalent field description in Section 9.11.1 (0x80 indicates that the field is not used)
- `u8DutyCycleApplied` is an optional field containing the percentage duty cycle applied by the sending client (if the user has chosen to over-ride the original setting in the LCE) - for the format of this setting, refer to the equivalent field description in Section 9.11.1 (0xFF indicates that the field is not used)
- `u8EventControl` is a bitmap which specifies whether a randomised start-time and/or randomised end-time are configured for the LCE:

Bit	Description
0	1 = randomised start-time, 0 = immediate start-time
1	1 = randomised end-time, 0 = immediate end-time
2-7	Not used

- `u8SignatureType` is the type of algorithm, if any, used to create the signature for the Report Event Status message (only one algorithm, ECDSA, is currently supported):

```
#define SE_DRLC_NO_SIGNATURE               (0x00)
#define SE_DRLC_SIGNATURE_TYPE_ECDSA       (0x01)
```

- u8CriticalityLevelApplied is the criticality level of the LCE - enumerations are provided and described in Section 9.10.3
- bSignatureVerified is filled in by the recipient of the Report Event Status message (therefore, the DRLC cluster server) to indicate whether the signature of the message has been verified and is valid:

  TRUE - verified and valid
  FALSE - verified and not valid, or not verified
- u16CoolingTemperatureSetPointApplied is an optional field containing the cooling temperature applied by the sending client (if the user has chosen to over-ride the original setting in the LCE) - for the format of this setting, refer to the equivalent field description in Section 9.11.1 (0x8000 indicates that the field is not used)
- u16HeatingTemperatureSetPointApplied is an optional field containing the heating temperature applied by the sending client (if the user has chosen to over-ride the original setting in the LCE) - for the format of this setting, refer to the equivalent field description in Section 9.11.1 (0x8000 indicates that the field is not used)
- u32IssuerId is the unique identifier for the LCE, as issued by the utility company
- u32EventStatusTime is the time (UTC) at which the Report Event Status message was issued
- sSignature is the signature for the Report Event Status message - this is the concatenation of two ECDSA signature components (r,s)

Note: It is recommended that signatures are supported by your application for backward compatibility.

9.11.5 tsSE_DRLCCallBackMessage

The structure of type tsSE_DRLCCallBackMessage contains a DRLC callback event. It is shown below but described in Section 9.7.

```
typedef struct
{
    teSE_DRLCCallBackEventType              eEventType;
    uint8                                   u8CommandId;
    teSE_DRLCStatus                         eDRLCStatus;
    uint32                                  u32CurrentTime;
    union {
        tsSE_DRLCLoadControlEvent           sLoadControlEvent;
        tsSE_DRLCCancelLoadControlEvent     sCancelLoadControlEvent;
        tsSE_DRLCCancelLoadControlAllEvent  sCancelLoadControlAllEvent;
        tsSE_DRLCReportEvent                sReportEvent;
        tsSE_DRLCGetScheduledEvents         sGetScheduledEvents;
    } uMessage;
} tsSE_DRLCCallBackMessage;
```

9.12 Compile-Time Options

This section describes the compile-time options that may be enabled in the **zcl_options.h** file of an application that uses the DRLC cluster.

The DRLC cluster is enabled by defining CLD_DRLC.

Client and server versions of the cluster are defined by DRLC_CLIENT and DRLC_SERVER, respectively.

Length of LCE Lists

The number of LCEs that may be stored in an LCE list (see Section 9.4.2) is, by default, three. This default can be over-ridden on the cluster server and a cluster client by assigning the desired values to the macros:

SE_DRLC_NUMBER_OF_SERVER_LOAD_CONTROL_ENTRIES (server)

SE_DRLC_NUMBER_OF_CLIENT_LOAD_CONTROL_ENTRIES (client)

LCE Re-sends

The DRLC cluster server may re-send an LCE when it becomes active in order to support clients that do not have a clock. This facility should not be enabled unless explicitly required. To enable this functionality, define:

DRLC_SEND_LCE_AGAIN_AT_ACTIVE_TIME

Message Signing (Security)

On DRLC cluster clients that need to implement message signing (see Section 9.6), the following must be defined:

```
#define SE_MESSAGE_SIGNING
```

For a DRLC cluster server to check a message signature, it is necessary to locally store the certificates of any nodes that perform key establishment. The maximum number of certificates that can be stored is configured by defining the following on the server:

```
#define KEC_NUM_CERTIFICATES <n>
```

where n is the number of certificates that can be stored.

10. Key Establishment Cluster

This chapter outlines the Key Establishment cluster which is defined in the ZigBee Smart Energy profile and is a mandatory cluster for nearly all ZigBee SE devices. It is used to manage secure wireless communications between SE devices.

The Key Establishment cluster has a Cluster ID of 0x0800.

10.1 Overview

The Key Establishment cluster is a mandatory server-side and client-side cluster for all ZigBee SE devices.

ZigBee PRO employs the AES-CCM* 128-bit key-based encryption system to secure communications between the nodes of a wireless network. The Key Establishment cluster is concerned with establishing a unique application link key for encrypted communications between a pair of nodes (the key is unique to the pair), as outlined in Section 2.5.2.

The attributes of the Key Establishment cluster client and server specify all the cryptographic schemes for key establishment on a device. The cluster's commands are used to initiate the key request mechanism on the client device. The ESP normally acts as the server and performs device authentication.

The Key Establishment cluster is enabled by defining CLD_KEY_ESTABLISHMENT in the **zcl_options.h** file - see Section 3.5.1. Further compile-time options for the Key Establishment cluster are detailed in Section 10.11.

Note: There can only be one Key Establishment cluster on each endpoint. The function **eZCL_Register()** checks for this and will return an error code of E_ZCL_ERR_KEY_ESTABLISHMENT_MORE_THAN_ONE_CLUSTER if more than one cluster is found on a single endpoint.

10.2 Key Establishment Cluster Structure and Attribute

The Key Establishment cluster is contained in the following `tsCLD_KeyEstablishment` structure:

```
typedef struct
{
    zenum16    u16KeyEstablishmentSuite;
} tsCLD_KeyEstablishment;
```

where `u16KeyEstablishmentSuite` indicates the Key Establishment suite used to derive the application link key. In the NXP implementation, this value should always be set to 1.

10.3 Performing Key Establishment

Key establishment is performed when a device joins the SE network and involves key exchanges with the ESP/Co-ordinator/Trust Centre - see Section 2.5.2. The process is largely automatic, but the applications on the joining device (client) and ESP (server) must take certain steps - these are detailed separately below for the joining device and ESP.

Note 1: The key establishment processes described below are illustrated in the demonstration code of the Application Note *Smart Energy HAN Solutions (JN-AN-1135)*.

Note 2: These processes apply on a cold start of the device or on a device reset in which the persisted application state indicates that key establishment did not previously complete (before the reset).

Note 3: In the case of a device reset in which the persisted application state indicates that key establishment successully completed (before the reset), security is re-established as described in Section 10.5.

On the Joining Device

The following steps must be taken in the application on the joining device:

1. First initialise the security set-up and register the pre-configured link key of the joining device (see Section 2.5.2) using the stack function **ZPS_vAplSecSetInitialSecurityState()** detailed in the *ZigBee PRO Stack User Guide (JN-UG-3048)*. Note that this function must be called after **ZPS_eAplAfInit()** and the JenOS initialisation functions that are listed in the “Forming a Network” section of the above User Guide.

2. Next initialise the SE library using **eSE_Initialise()** and register an endpoint on the joining device, e.g. using **eSE_RegisterIPDEndPoint()** for an IPD - see Section 4.2.
3. Now initialise the key establishment data by calling the SE function **eSE_KECLoadKeys()**. The joining device's digital security certificate, together with its associated public and private keys, must be provided in this function call (see Section 2.5.2).
4. At this point, the ZigBee PRO stack can be started on the joining device using the stack function **ZPS_eAplZdoStartStack()** and then the stack function **ZPS_eAplZdoJoinNetwork()** can be called to request a join. Note that full descriptions of initialising the application, starting the stack and initiating a join are provided in the *ZigBee PRO Stack User Guide (JN-UG-3048)*.
5. On receiving a confirmation event that the device has joined the network (depending on the device type, ZPS_EVENT_NWK_JOINED_AS_ROUTER or ZPS_EVENT_NWK_JOINED_AS_ENDDEVICE):
   - The network address and endpoint number of the Key Establishment cluster server should be discovered using the stack function **ZPS_eAplZdpMatchDescRequest()**, as described in Section 4.4.
   - The generation of APS acknowledgements must be disabled (until after key establishment has completed) using the function **ZPS_eAplZdoSetDevicePermission()** with the setting ZPS_DEVICE_PERMISSIONS_DATA_REQUEST_DISALLOWED.
6. The key exchange sequence can then be started by calling the SE function **eSE_KECInitiateKeyEstablishment()**. The Key Establishment cluster code then works through a pre-ordered sequence of exchanges with the server.
7. Eventually, a Key Establishment event will be generated indicating success or failure (for details of the Key Establishment events, see Section 10.4):
   - In the case of success, the function **ZPS_eAplZdoAddReplaceLinkKey()** must be called again to store the established application link key delivered in the `tsKEC_Common` structure in the generated event. Subsequent data exchanges between the joined device and the ESP will be encrypted with this new key. These communications should always use the joined device's IEEE/MAC address extracted from the device's security certificate in the `tsKEC_Common` structure. APS acknowledgements should also be re-enabled by calling **ZPS_eAplZdoSetDevicePermission()** with the setting ZPS_DEVICE_PERMISSIONS_ALL_PERMITED.
   - In the case of failure, the application may report the failure to the user, giving the option to re-try key establishment on the same device or start a channel scan for another ESP/Co-ordinator. Failure can occur if the ESP is busy with another key establishment process - in this case, the event will contain a minimum waiting time (in seconds) before a re-try is recommended.

On the ESP

The following steps must be taken in the application on the ESP/Co-ordinator:

Note: This procedure assumes that the ZigBee PRO stack is already running on the ESP/Co-ordinator and that the device has been fully initialised.

1. After registering the ESP endpoint using **eSE_RegisterEspMeterEndPoint()** or **eSE_RegisterEspEndPoint()**, initialise the key establishment data of the ESP device by calling the SE function **eSE_KECLoadKeys()**. The ESP device's digital security certificate, together with its associated public and private keys, must be provided in this function call (see Section 2.5.2).
2. For each device which is expected to join the network, perform the following two steps:
   a) Load the pre-configured link key of the joining device, using the stack function **ZPS_eAplZdoAddReplaceLinkKey()** detailed in the *ZigBee PRO Stack User Guide (JN-UG-3048)*. The pre-configured link key and IEEE/MAC address of the joining node are provided to the ESP application by the utility company via the backhaul network - see Section 2.5.2.
   b) Call **ZPS_bAplZdoTrustCenterSetDevicePermissions()** with the setting ZPS_TRUST_CENTER_DATA_REQUEST_DISALLOWED for the joining node.
3. On completion of the key exchange sequence with a joining device, a Key Establishment event will be generated indicating success or failure (for details of the Key Establishment events, see Section 10.4):
   - In the case of success, the function **ZPS_eAplZdoAddReplaceLinkKey()** must be called again to store the established application link key (delivered in the generated event). The ESP application must then call the stack function **ZPS_bAplZdoTrustCenterSetDevicePermissions()** with the setting ZPS_TRUST_CENTER_JOIN_DISALLOWED for the stored key (so that the device cannot ‘join’ the network again but only ‘re-join’). Subsequent data exchanges between the ESP and the joined device will be encrypted with this key.
   - In the case of failure (which can occur if the ESP is busy with another key establishment process), it is the responsibility of the client (joined device) to decide what to do next, e.g. re-try key establishment after a short delay.

10.4 Key Establishment Events

The Key Establishment cluster has its own events that are handled through the callback mechanism outlined in Section 4.7 (and fully detailed in the *ZCL User Guide (JN-UG-3077)*). Key Establishment event handling must be included in the callback function for the associated endpoint, where this callback function is registered through the relevant endpoint registration function (for example, through **eSE_RegisterEspEndPoint()** for a standalone ESP). The relevant callback function will then be invoked when a Key Establishment event occurs.

For a Key Establishment event, the `eEventType` field of the `tsZCL_CallBackEvent` structure is set to E_ZCL_CBET_CLUSTER_CUSTOM. This event structure also contains an element `sClusterCustomMessage`, which is itself a structure containing a field `pvCustomData`. This field is a pointer to a `tsSE_KECCallBackMessage` structure which contains the Key Establishment parameters:

```
typedef struct
{
    teSE_KECCallBackEventType  eEventType;
    uint8                      u8CommandId;
    teSE_KECStatus             eKECStatus;
    teSE_KECTerminateKeyEstablishmentStatusCode eTerminateReason;
    uint32                     u32CurrentTime;
    tsKEC_Common               *psKEC_Common;
} tsSE_KECCallBackMessage;
```

Information on the elements of the above structure is provided in the sub-sections below. The structure is fully detailed in Section 10.9.1.

10.4.1 Event Types

The eEventType field of the tsSE_KECCallBackMessage structure above specifies the type of Key Establishment event that has been generated - these event types are enumerated in the teSE_KECCallBackEventType structure and described below.

```
typedef enum PACK
{
    E_SE_KEC_EVENT_API =0x00,
    E_SE_KEC_EVENT_COMMAND,
    // scheduler codes
    E_SE_KEC_APS_RX_EXPIRED,
    E_SE_KEC_EPHEMERAL_DATA_EXPIRED,
    E_SE_KEC_KEY_ESTABLISHMENT_EXPIRED,
    E_SE_KEC_KEY_ESTABLISHMENT_TERMINATED
} teSE_KECCallBackEventType;
```

E_SE_KEC_EVENT_COMMAND

The E_SE_KEC_EVENT_COMMAND event is generated when a command is executing on either the server or client. In the tsSE_KECCallBackMessage structure, the u8CommandId field is used to indicate the corresponding command - one of:

```
#define E_SE_INITIATE_KEY_ESTABLISHMENT_REQUEST    (0x00)
#define E_SE_EPHEMERAL_DATA_REQUEST    (0x01)
#define E_SE_CONFIRM_KEY_DATA_REQUEST    (0x02)
#define E_SE_TERMINATE_KEY_ESTABLISHMENT    (0x03)

#define E_SE_INITIATE_KEY_ESTABLISHMENT_RESPONSE   (0x00)
#define E_SE_EPHEMERAL_DATA_RESPONSE    (0x01)
#define E_SE_CONFIRM_KEY_DATA_RESPONSE    (0x02)
```

E_SE_KEC_APS_RX_EXPIRED

The E_SE_KEC_APS_RX_EXPIRED event is generated when the key establishment process has timed out due to an acknowledgement not being received in response to the last key establishment message that was transmitted.

E_SE_KEC_EPHEMERAL_DATA_EXPIRED

The E_SE_KEC_EPHEMERAL_DATA_EXPIRED event is generated when the key establishment process has timed out due to ephemeral data not being generated in a given time.

E_SE_KEC_KEY_ESTABLISHMENT_EXPIRED

The E_SE_KEC_KEY_ESTABLISHMENT_EXPIRED event is generated when the key establishment process has timed out due to the Confirm Key request not being generated in time by the client.

E_SE_KEC_KEY_ESTABLISHMENT_TERMINATED

The E_SE_KEC_KEY_ESTABLISHMENT_TERMINATED event is generated when the key establishment process has been terminated prematurely (for one of a number of reasons - see Section 10.10.3).

10.4.2 Other Elements of tsSE_KECCallBackMessage

In addition to the fields `eEventType` and `u8CommandId` described in Section 10.4.2, the `tsSE_KECCallBackMessage` structure contains the following elements.

u32CurrentTime

The `u32CurrentTime` field contains the time (UTC) at which the event was generated.

psKEC_Common

The `psKEC_Common` field is a pointer to the internal key exchange data structure, which defines the present state of the Key Establishment cluster. The user should not modify this data.

eKECStatus

The `eKECStatus` field indicates the status returned from the command that has been executed (the command identified in `u8CommandId`). The status codes are enumerated in the `teSE_KECStatus` structure, shown below.

```
typedef enum PACK
{
    E_SE_KEY_ESTABLISHMENT_SUCCESS =0x00,
    E_SE_KEY_ESTABLISHMENT_FAILURE,
    E_SE_KEY_ESTABLISHMENT_VALUE,
    E_SE_KEY_ESTABLISHMENT_PARAMETER_NULL,
    E_SE_KEY_ESTABLISHMENT_PARAMETER_ERROR,
    E_SE_KEY_ESTABLISHMENT_INSUFFICIENT_SPACE,
    E_SE_KEY_ESTABLISHMENT_STATE_ERROR,
    E_SE_KEY_ESTABLISHMENT_EP_NOT_FOUND,
    E_SE_KEY_ESTABLISHMENT_EP_RANGE,
    E_SE_KEY_ESTABLISHMENT_ZBUFFER_FAIL,
    E_SE_KEY_ESTABLISHMENT_CLUSTER_NOT_FOUND,
    E_SE_KEY_ESTABLISHMENT_CUSTOM_DATA_NULL,
    E_SE_KEY_ESTABLISHMENT_ZTRANSMIT_FAIL,
    E_SE_KEY_ESTABLISHMENT_BAD_MESSAGE,
    E_SE_KEY_ESTABLISHMENT_NO_RESOURCES,
    E_SE_KEY_ESTABLISHMENT_UNSUPPORTED_SUITE,
    E_SE_KEY_ESTABLISHMENT_CERTIFICATE_ERROR,
    E_SE_KEY_ESTABLISHMENT_TSN_ERROR
} teSE_KECStatus;
```

10.5 Restoring Link Key from Non-Volatile Memory

An application link key (which results from the key establishment process) is stored in external non-volatile memory (normally Flash memory) when the function **ZPS_eAplZdoAddReplaceLinkKey()** is called. During a subsequent cold restart of the JN51xx wireless microcontroller (for example, following a power failure), this key is automatically retrieved from Flash memory, provided that the JenOS Persistent Data Manager (PDM) module is enabled - for details of using the PDM module, refer to the *JenOS User Guide (JN-UG-3075)*.

In order to successfully restore the application link key from Flash memory, you should implement the following procedure in your application (which requires use of the PDM module) - the exact procedure required depends on the device:

Before the Restart

1. Immediately after calling **ZPS_eAplZdoAddReplaceLinkKey()** to store the newly established key, save a variable to Flash memory indicating that key establishment has completed and an application link key has been stored.
2. Save this variable to Flash memory using the PDM function **PDM_vSaveRecord()**.

After the Restart (ESP/Trust Centre)

1. During a subsequent cold start, retrieve the saved variable from Flash memory using the PDM function **PDM_eLoadRecord()** and check the variable to confirm that an application link key is available.
2. Provided that a key is available, call the following functions in your code:
   - **a)** **eSE_Initialise()** to initialise the SE library
   - **b)** Relevant endpoint registration function, e.g. **eSE_RegisterEspEndPoint()**
   - **c)** **ZPS_eAplZdoStartStack()** to start the stack

After the Restart (Other Devices)

1. During a subsequent cold start, retrieve the saved variable from Flash memory using the PDM function **PDM_eLoadRecord()** and check the variable to confirm that an application link key is available.
2. Provided that a key is available, call the following functions in your code:
   - **a)** **ZPS_eAplZdoStartStack()** to start the stack
   - **b)** **eSE_Initialise()** to initialise the SE library
   - **c)** Relevant endpoint registration function, e.g. **eSE_RegisterIPDEndPoint()**
   - **d)** **ZPS_eAplZdoSetDevicePermission()** with the setting ZPS_DEVICE_PERMISSIONS_ALL_PERMITED to re-enable APS acknowledgements

***Caution:** If an established application link key has been restored from Flash memory, do not call the function **ZPS_eAplZdoAddReplaceLinkKey()**, as this will overwrite the existing key.*

If performing a cold start and there is no application link key to retrieve from Flash memory, the application must initiate a new key establishment process, as described in Section 10.3.

10.6 Testing Key Establishment

The function **eSE_KECConfigureTestHarness()** allows certain tests on key establishment between devices to be configured on a device. The parameter values for this configuration are provided to the function in a structure of the type **tsKEC_TestHarnessParameters** (described in Section 10.9.3). The possible tests and the relevant parameters from this structure are summarised in the table below.

Test	Description	Relevant Parameters
Timeout (Trust Centre)	Tests the Trust Centre's handling of a timeout when the partner device takes longer than advertised to respond to a key establishment cluster message	`bSlowResponseTestEnabled` `u8EphemeralInitMessageTime` `u8EphemeralDelayTime`
Timeout (Device)	Tests a device's handling of a timeout when the partner device takes longer than advertised to respond to a key establishment cluster message	
Long Generate Time (Trust Centre)	Tests the Trust Centre's handling of large values of Ephemeral Data Generate Time and Confirm Key Generate Time when the partner device meets these requirements	`bSlowResponseTestEnabled` `u8EphemeralInitMessageTime` `u8EphemeralDelayTime` `u8ConfirmInitMessageTime` `u8ConfirmDelayTime`
Long Generate Time (Device)	Tests a device's handling of large values of Ephemeral Data Generate Time and Confirm Key Generate Time when the partner device meets these requirements	
Too Long Certificate (Trust Centre)	Tests the proper operation of the Trust Centre when it receives a key establishment message which has extra bytes appended to the message	`u8ExtraBytesAfterCert`
Too Long Certificate (Device)	Tests the proper operation of a device when it receives a key establishment message which has extra bytes appended to the end of the message	
Out of Sequence Message (Device)	Tests the proper handling of an out-of-order message in the key establishment protocol	`bRespondToInitReqWithEp-hRsp`
Corrupt Certificate (Device)	Tests the rejection of link key establishment between the Trust Centre and a device due to a corrupted certificate	

Table 42: Key Establishment Tests

For full details of the above tests, refer to the "Security & The Key Establishment Cluster" section of the *SEP 1.1 Test Specification (075384)* from the ZigBee Alliance.

10.7 Functions

The following Key Establishment cluster functions are provided in the SE API:

Function	Page
eSE_KECCreate	288
eSE_KECLoadKeys	290
eSE_KECInitiateKeyEstablishment	291
eSE_KECConfigureTestHarness	292

eSE_KECCreate

teZCL_Status eSE_KECCreate(
  uint8 ***pu8AttributeControlBits,**
  tsZCL_ClusterInstance ***psClusterInstance,**
  tsZCL_ClusterDefinition ***psClusterDefinition,**
  tsSE_KECCustomDataStructure ***psCustomDataStructure,**
  void ***pvEndPointSharedStructPtr);**

Description

This function creates an instance of the Key Establishment cluster on an endpoint. The cluster instance is created on the endpoint which is associated with the supplied `tsZCL_ClusterInstance` structure.

The function should only be called when setting up a custom endpoint containing one or more selected clusters (rather than the whole set of clusters supported by a standard ZigBee device). This function will create a Key Establishment cluster instance on the endpoint, but instances of other clusters may also be created on the same endpoint by calling their corresponding creation functions. For more details of creating cluster instances on custom endpoints, refer to Appendix B.

Note: This function must not be called for an endpoint on which a standard ZigBee device (e.g. IPD) will be used. In this case, the device and its supported clusters must be registered on the endpoint using the relevant device registration function from those described in Chapter 12.

When used, this function must be the first Key Establishment cluster function called in the application, and must be called after the stack has been started and after the application profile has been initialised.

The function requires an array to be declared for internal use, which contains one element (of type **uint8**) for each attribute of the cluster. The array length should therefore equate to the total number of attributes supported by the Key Establishment cluster, which can be obtained by using the macro CLD_KEC_MAX_NUMBER_OF_ATTRIBUTE.

The array declaration should be as follows:

```
uint8 au8AppKEC_ClusterAttributeControlBits[CLD_KEC_MAX_NUMBER_OF_ATTRIBUTE];
```

The function will initialise the array elements to zero.

Parameters

pu8AttributeControlBits	Pointer to an array of **uint8** values, with one element for each attribute in the cluster (see above).
psClusterInstance	Pointer to structure containing information about the cluster instance to be created (see the *ZCL User Guide (JN-UG-3077)*). This structure will be updated by the function by initialising individual structure fields.

psClusterDefinition	Pointer to structure indicating the type of cluster to be created (see the *ZCL User Guide (JN-UG-3077)*). In this case, this structure must contain the details of the Key Establishment cluster. This parameter can refer to a pre-filled structure called `sCLD_KeyEstablishment` which is provided in the **KeyEstablishment.h** file.
psCustomDataStructure	Pointer to structure which contains custom data for the Key Establishment cluster. This structure is used for internal data storage. No knowledge of the fields of this structure is required
pvEndPointSharedStructPtr	Pointer to the shared structure used for attribute storage. This parameter should be the address of the structure of type `tsCLD_KeyEstablishment` which defines the attributes of Key Establishment cluster. The function will initialise the attributes with default values.

Returns

E_ZCL_SUCCESS

E_ZCL_FAIL

E_ZCL_ERR_PARAMETER_NULL

E_ZCL_ERR_INVALID_VALUE

eSE_KECLoadKeys

```
teSE_KECStatus eSE_KECLoadKeys(
                    uint8 u8SourceEndPointId,
                    uint8 *pu8CertificateAuthorityPublicKey,
                    uint8 *pu8Certificate,
                    uint8 *pu8PrivateKey);
```

Description

This function must be called by the application on both the joining device and the ESP, in order to load the respective device's security certificate and keys. The function can be called once the device endpoint has been registered, e.g. after calling **eSE_RegisterEspEndPoint()** for a stanadlone ESP or **eSE_RegisterIPDEndPoint()** for an IPD.

For more information on where to use this function, refer to Section 10.3.

A digital security certificate must be obtained for a device from Certicom (www.certicom.com). This certificate includes a public key. A private key is also issued, which is paired with the certificate. The security certificate, public key and private key must all be passed into this function.

Note that the Key Establishment cluster code makes internal copies of the data passed via this function and there is therefore no need for the application to maintain this data as part of external context saving.

Parameters

u8SourceEndPointId	Number of the local endpoint on which the Key Establishment cluster resides
pu8CertificateAuthorityPublicKey	Pointer to location which holds the device's public key, issued as part of its security certificate
pu8Certificate	Pointer to location which holds the device's security certificate
pu8PrivateKey	Pointer to location which holds the device's private key, issued alongside its security certificate

Returns

E_ZCL_SUCCESS

E_ZCL_ERR_PARAMETER_NULL

eSE_KECInitiateKeyEstablishment

```
teSE_KECStatus eSE_KECInitiateKeyEstablishment(
                    uint8 u8SourceEndPointId,
                    uint8 u8DestinationEndPointId,
                    tsZCL_Address *psDestinationAddress,
                    uint8 *pu8TransactionSequenceNumber);
```

Description

This function must be called by the application on a joined device to initiate the key exchange protocol with the ESP. The function must be called only once the function **eSE_KECLoadKeys()** has been called on both the local device and ESP, and once the stack has been started on the local device and the device has joined the network.

On successful completion of the key establishment process, an application link key will be returned in an E_ZCL_CBET_CLUSTER_CUSTOM event of the type E_SE_KEC_EVENT_COMMAND. This key must subsequently be stored by the application using the ZigBee PRO function **ZPS_eAplZdoAddReplaceLinkKey()**.

For more information on where to use this function, refer to Section 10.3.

Parameters

u8SourceEndPointId	Number of the local endpoint through which the key exchange will take place
u8DestinationEndPointId	Number of the remote endpoint (on the ESP) to which key exchange messages will be sent
psDestinationAddress	Pointer to a structure containing the address of the remote node (ESP) to which key exchange messages will be sent
pu8TransactionSequenceNumber	Pointer to a location to store the Transaction Sequence Number (TSN) of the request

Returns

E_ZCL_SUCCESS

E_ZCL_ERR_PARAMETER_NULL

E_SE_KEY_ESTABLISHMENT_STATE_ERROR

E_SE_KEY_ESTABLISHMENT_KEYS_NOT_LOADED

eSE_KECConfigureTestHarness

```
teSE_KECStatus eSE_KECConfigureTestHarness(
                uint8 u8SourceEndPointId,
                tsKEC_TestHarnessParameters *psParameters);
```

Description

This function can be used to set up a test harness in order to perform tests on key establishment between two devices, one of which is the Trust Centre. The function is used to configure certain key establishment parameters on one of the two participating devices. The parameter values that define the particular test(s) to be performed are provided through a structure (detailed in Section 10.9.3).

The possible tests are outlined in Section 10.6. For full details of the tests, refer to the "Security & The Key Establishment Cluster" section of the *SEP 1.1 Test Specification (075384)* from the ZigBee Alliance.

Caution: *This function should only be called during testing. Be sure to remove the function call for normal operation of the network.*

Parameters

u8SourceEndPointId	Number of the local endpoint through which the key exchange will take place
psParameters	Pointer to structure containing parameter values for test harness (see Section 10.9.3)

Returns

E_ZCL_SUCCESS
E_ZCL_ERR_EP_RANGE
E_ZCL_ERR_CLUSTER_NOT_FOUND
E_ZCL_ERR_CUSTOM_DATA_NULL

10.8 Return Codes

In addition to some of the ZCL status enumerations (detailed in the *ZCL User Guide (JN-UG-3077)*), the following enumerations are returned by SE API Key Establishment cluster functions (see Section 10.7) to indicate the outcome of the function call.

```
typedef enum PACK
{
    E_SE_KEY_ESTABLISHMENT_STATE_ERROR = 0x80,
    E_SE_KEY_ESTABLISHMENT_BAD_MESSAGE,
    E_SE_KEY_ESTABLISHMENT_NO_RESOURCES,
    E_SE_KEY_ESTABLISHMENT_UNSUPPORTED_SUITE,
    E_SE_KEY_ESTABLISHMENT_TSN_ERROR,
    E_SE_KEY_ESTABLISHMENT_MAC_VERIFY_FAILED,
    E_SE_KEY_ESTABLISHMENT_KEYS_NOT_LOADED
} teSE_KECStatus;
```

The above enumerations are described in the table below.

Enumeration	Description
E_SE_KEY_ESTABLISHMENT_STATE_ERROR	Error occurred during key exchange process
E_SE_KEY_ESTABLISHMENT_BAD_MESSAGE	Incorrect parameter in key exchange message
E_SE_KEY_ESTABLISHMENT_NO_RESOURCES	Key establishment currently busy
E_SE_KEY_ESTABLISHMENT_UNSUPPORTED_SUITE	Specified Key Establishment suite is not supported
E_SE_KEY_ESTABLISHMENT_TSN_ERROR	Error in Transaction Sequence Number
E_SE_KEY_ESTABLISHMENT_MAC_VERIFY_FAILED	Message Authentication Code incorrect
E_SE_KEY_ESTABLISHMENT_KEYS_NOT_LOADED	No keys have been loaded

Table 43: Key Establishment Cluster Return Codes

10.9 Structures

10.9.1 tsSE_KECCallBackMessage

For a Key Establishment event, the `eEventType` field of the `tsZCL_CallBackEvent` structure is set to E_ZCL_CBET_CLUSTER_CUSTOM. This event structure also contains an element `sClusterCustomMessage`, which is itself a structure containing a field `pvCustomData`. This field is a pointer to the following `tsSE_KECCallBackMessage` structure which contains the Key Establishment parameters:

```
typedef struct
{
    teSE_KECCallBackEventType   eEventType;
    uint8                       u8CommandId;
    teSE_KECStatus              eKECStatus;
    teSE_KECTerminateKeyEstablishmentStatusCode eTerminateReason;
    uint32                      u32CurrentTime;
    tsKEC_Common                *psKEC_Common;
} tsSE_KECCallBackMessage;
```

where:

- `eEventType` is the Key Establishment event type - see Section 10.9
- `u8CommandId` is the command that has been executed - see Section 10.10
- `eKECStatus` is the return status of the command that has been executed - see Section 10.8
- `eTerminateReason` is an enumeration indicating the reason for terminating the key establishment process - see Section 10.10.3
- `u32CurrentTime` is the current time (UTC), in seconds
- `psKEC_Common` is a pointer to the internal key exchange data structure, which defines the present state of the Key Establishment cluster (this structure contains the established application link key) - see Section 10.9.2

10.9.2 tsKEC_Common

This structure contains information relating to the key establishment process, but most of this information is not directly relevant to the application. The structure is accessed via the `tsSE_KECCallBackMessage` structure (see Section 10.9.1) and contains the application link key which results from a successful key establishment process.

```
struct tsKEC_Common
{
    uint8                        u8RemoteEmphemeralTimeout;
    uint8                        u8RemoteConfirmTimeout;
    uint8                        u8WaitTime;
    uint8                        u8TransactionSequenceNumber;
    bool_t                       bAPSrxReceived;
    bool_t                       bIsLoaded;
    teSE_KECState                eState;
    uint32                       u32APSRxUtcTimeout;
    uint32                       u32NextMessageUtcTimeout;
    tsZCL_CallBackEvent          sKECCustomCallBackEvent;
    tsSE_KECCallBackMessage      sKECCallBackMessage;
    tsKEC_TestHarnessParameters   sTestHarnessParameters;
    tsKEC_TestHarnessState        sTestHarnessState;
    tsSE_KECTerminateKeyEstablishmentCmdPayload sTerminateKeyEstablishmentCmdPayload;
    uint8                        au8LocalEphemeralPublicKey[E_SE_PUBLIC_KEY_LEN];
    uint8                        au8LocalMACVData[E_SE_HASH_LEN];
    union
    {
        uint8                    au8RemoteEphemeralPublicKey[E_SE_PUBLIC_KEY_LEN];
        uint8                    au8RemoteCertificate[E_SE_CERTIFICATE_LEN];
    }uMessage;
    uint8                        *pu8CertificateAuthorityPublicKey;
    uint8                        *pu8LocalCertificate;
    uint8                        *pu8PrivateKey;
    uint8                        au8EphemeralPrivateKey[E_SE_PRIVATE_KEY_LEN];
    uint8                        au8MacKey[E_SE_HASH_LEN];
    uint8                        au8Key[E_SE_HASH_LEN];
    tsSE_KECCertificate          asCertificate[KEC_NUM_CERTIFICATES];
};
```

The only elements of this structure that will be of interest to the application are:

- `au8RemoteCertificate[]`, which is an array containing the security certificate of the remote device with which a link key has been established - this certificate contains the IEEE/MAC address of the remote device (see below)
- `au8Key[]`, which is an array containing the established application link key

The IEEE/MAC address of the remote device and the application link key can be extracted from this structure using the following code:

```
// Copy MAC address from certificate
memcpy(&u64MacAddress, &psKECMessage->psKEC_Common->au8RemoteCertificate[22], 8);

ZPS_eAplZdoAddReplaceLinkKey(u64MacAddress, psKECMessage->psKEC_Common->au8Key);
```

10.9.3 tsKEC_TestHarnessParameters

This structure contains parameter values which define the test(s) to be conducted on key establishment between devices. These parameter values are used by the function **eSE_KECConfigureTestHarness()**.

```
typedef struct
{
    bool_t bSlowResponseTestEnabled;
    uint8 u8EphemeralDelayTime;
    uint8 u8ConfirmDelayTime;
    uint8 u8EphemeralInitMessageTime;
    uint8 u8ConfirmInitMessageTime;
    uint8 u8ExtraBytesAfterCert;
    bool_t bRespondToInitReqWithEphRsp;
} tsKEC_TestHarnessParameters;
```

where:

- `bSlowResponseTestEnabled` enables (TRUE) or disables (FALSE) the testing of slow message generation, controlled by the following parameters:
    - `u8EphemeralDelayTime` is the time delay, in seconds, to apply before sending an Ephemeral message (e.g. 240 s)
    - `u8ConfirmDelayTime` is the time delay, in seconds, to apply before sending a Confirm Key message (e.g. 240 s)
    - `u8EphemeralInitMessageTime` is the Ephemeral Delay Time, in seconds, advertised in the Initiate message (e.g. 254 s)
    - `u8ConfirmInitMessageTime` is the Confirm Generate Time, in seconds, advertised in the Initiate message (e.g. 254 s)
- `u8ExtraBytesAfterCert` is the number of bytes to insert after the identity data in the Initiate message
- `bRespondToInitReqWithEphRsp` enables (TRUE) or disables (FALSE) an Ephemeral Data Response to the Initiate message for testing out-of-order messages

For information on the possible tests and the relevant parameters (from the above structure), refer to Section 10.6.

10.10 Enumerations

10.10.1 'Event' Enumerations

The event types generated by the Key Establishment cluster are enumerated in the `teSE_KECCallBackEventType` structure below:

```
typedef enum PACK
{
   E_SE_KEC_EVENT_API =0x00,
   E_SE_KEC_EVENT_COMMAND,
   // scheduler codes
   E_SE_KEC_APS_RX_EXPIRED,
   E_SE_KEC_INITIATE_EXPIRED,
   E_SE_KEC_EPHEMERAL_DATA_EXPIRED,
   E_SE_KEC_KEY_ESTABLISHMENT_EXPIRED,
   E_SE_KEC_KEY_ESTABLISHMENT_TERMINATED,
   E_SE_KEC_KEY_ESTABLISHMENT_CBET_ENUM_END,
} teSE_KECCallBackEventType;
```

The above event types are described in the table below.

Event Type Enumeration	Description
E_SE_KEC_EVENT_API	Not used - reserved for future use
E_SE_KEC_EVENT_COMMAND	Indicates that a command is executing on the server or client
E_SE_KEC_APS_RX_EXPIRED	Indicates that the key establishment process has timed out due to an APS acknowledgement not being received in response to the last key establishment message that was transmitted
E_SE_KEC_INITIATE_EXPIRED	Indicates a timeout due to an 'initiate response' not being received
E_SE_KEC_EPHEMERAL_DATA_EXPIRED	Indicates that the key establishment process has timed out due to ephemeral data not being generated in a given time.
E_SE_KEC_KEY_ESTABLISHMENT_EXPIRED	Indicates that the key establishment process has timed out due to the Confirm Key request not being generated in time by the client
E_SE_KEC_KEY_ESTABLISHMENT_ TERMINATED	Indicates that key establishment with the server has been terminated due to a mismatch of server certificates

Table 44: Key Establishment Event Types

10.10.2 ‘Command ID’ Enumerations

The following enumerations are used to identify commands (requests and responses) used by the Key Establishment cluster:

```
#define E_SE_INITIATE_KEY_ESTABLISHMENT_REQUEST    (0x00)
#define E_SE_EPHEMERAL_DATA_REQUEST                (0x01)
#define E_SE_CONFIRM_KEY_DATA_REQUEST              (0x02)
#define E_SE_TERMINATE_KEY_ESTABLISHMENT           (0x03)

#define E_SE_INITIATE_KEY_ESTABLISHMENT_RESPONSE   (0x00)
#define E_SE_EPHEMERAL_DATA_RESPONSE               (0x01)
#define E_SE_CONFIRM_KEY_DATA_RESPONSE             (0x02)
```

On successful completion of the key establishment process, the response E_SE_CONFIRM_KEY_DATA_RESPONSE will be generated.

10.10.3 ‘Key Establishment Termination’ Status Codes

The following enumerations are used to indicate the reason for the termination of the key establishment process.

typedef enum

```
{
    E_SE_KEY_ESTABLISHMENT_TERMINATE_UNKNOWN_ISSUER = 0x01,
    E_SE_KEY_ESTABLISHMENT_TERMINATE_BAD_KEY_CONFIRM,
    E_SE_KEY_ESTABLISHMENT_TERMINATE_BAD_MESSAGE,
    E_SE_KEY_ESTABLISHMENT_TERMINATE_NO_RESOURCES,
    E_SE_KEY_ESTABLISHMENT_TERMINATE_UNSUPPORTED_SUITE,
} teSE_KECTerminateKeyEstablishmentStatusCode;
```

The above status codes are described in the table below.

Event Type Enumeration	Description
E_SE_KEY_ESTABLISHMENT_TERMINATE_UNKNOWN_ISSUER	Unknown issuer
E_SE_KEY_ESTABLISHMENT_TERMINATE_BAD_KEY_CONFIRM	Bad key
E_SE_KEY_ESTABLISHMENT_TERMINATE_BAD_MESSAGE	Bad message
E_SE_KEY_ESTABLISHMENT_TERMINATE_NO_RESOURCES	No resources to handle key establishment

Table 45: ‘Key Establishment Termination’ Status Codes

Event Type Enumeration	Description
E_SE_KEY_ESTABLISHMENT_TERMINATE_ UNSUPPORTED_SUITE	Unsupported KEC suite

Table 45: 'Key Establishment Termination' Status Codes

10.11 Compile-Time Options

This section describes the compile-time options that may be set in the **zcl_options.h** file of an application that uses the Key Establishment cluster.

The Key Establishment cluster is enabled by defining CLD_KEC.

Tip: The Itron meter uses its own ZCL Transaction Sequence Number (TSN) when replying to messages, rather than the TSN of the message that it is replying to, and so the following option must be defined: CLD_KEY_ESTABLISHMENT_DISABLE_TSN_CHECK.

A set of key establishment timeout periods can be customised in the options file, although it is unlikely that you will need to change these from the default values (which are defined in the header file **KEC.h**). These timeout values are described below.

Ephemeral Data Timeout

The timeout period for the generation of an ephemeral data message is defined using the macro:

KEC_EPHEMERAL_DATA_GENERATE_TIME

The default value is 10 seconds (which is conservative for the JN51xx device, which normally takes less than one second to generate this message). This value is sent to the remote device to inform it how long to wait for an ephemeral data message before timing out.

'Confirm Key' Timeout

The timeout period for the generation of a 'confirm key' message is defined using the macro:

KEC_CONFIRM_KEY_GENERATE_TIME

The default value is 10 seconds (which is conservative for the JN51xx device, which normally takes less than one second to generate this message). This value is sent to the remote device to inform it how long to wait for a key confirmation message before timing out.

Message Response Timeout

The timeout period for the reception of the response to a key establishment message is defined using the macro:

KEC_MESSAGE_RESPONSE_TIME

The default value is 7 seconds (which takes into account possible transmission retries and is comfortably above the minimum of 2 seconds recommended in the ZigBee Smart Energy Specification). This value is added to the timeout value for the particular message type - for example, when a JN51xx device performs key establishment with another JN51xx device, the total message-response timeout period for an ephemeral data message or a ‘confirm key’ message is 17 seconds (by default).

APS Acknowledgement Timeout

The timeout period for the generation of an APS acknowledgement is defined using the macro:

KEC_APS_ACK_WAIT_TIME

By default, this value is set to the value of KEC_MESSAGE_RESPONSE_TIME.

‘Initiate Response’ Timeout

The timeout period for the generation of an ‘initiate response’ is defined using the macro:

KEC_INITIATE_RESPONSE_WAIT_TIME

By default, this value is set to the value of KEC_MESSAGE_RESPONSE_TIME. The value is used locally after sending an ‘initiate message’ and while waiting for the response.

Private Key Encryption

The following macro must be enabled for any JN516x production applications that use the Key Establishment cluster:

KEC_DECRYPT_PRIVATE_KEY

This macro enables decryption of the private security key. The key should therefore be encrypted before it is merged with the application binary using JET. For more information, refer to the *JET User Guide (JN-UG-3081)*.

11. Tunnelling Cluster

This chapter outlines the Tunnelling cluster which is defined in the ZigBee Smart Energy profile. It provides a transport mechanism for metering protocols within the payloads of standard ZigBee frames.

The Tunnelling cluster has a Cluster ID of 0x0704.

Note: The NXP implementation of the Tunnelling cluster does not currently support the optional flow control feature.

11.1 Overview

Metering data, which uses its own transfer protocol, can be transported through an SE network by embedding the data directly inside the payloads of normal ZigBee frames. This data is said to be 'tunnelled' through the network. For example, data utilising the following metering transfer protocols can be tunnelled: DLMS/COSEM, IEC61107, ANSI C12, M-Bus. This functionality is implemented using the Tunnelling cluster.

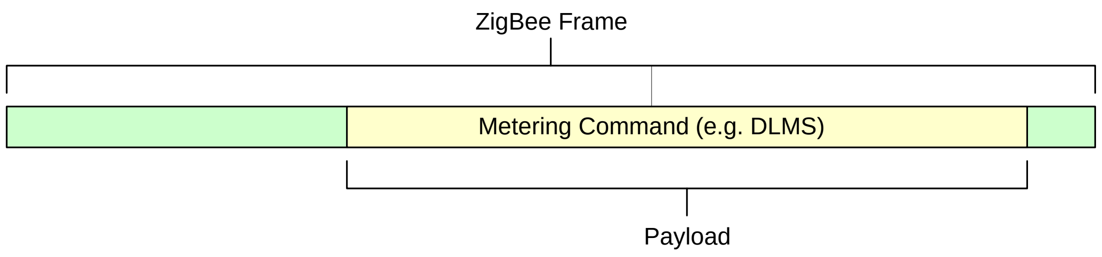

Figure 16: Tunnelling Metering Data in ZigBee Frames

Figure 17 below illustrates how the Tunnelling cluster may be used to transport DLMS protocol messages within an SE network. The network includes a DLMS station (located at the utility company) which sends commands to a Metering Device supporting DLMS in the HAN at a customer's premises. The ESP of the HAN receives a DLMS command from the DLMS station via the backhaul network. The ESP transparently tunnels the DLMS command to the Metering Device by encapsulating the command in the payload of a ZigBee frame. The Metering Device receives the tunnelled command, extracts and processes the DLMS message in the payload, creates a response message and tunnels the DLMS response message back to the ESP, which passes the data payload to the DLMS station.

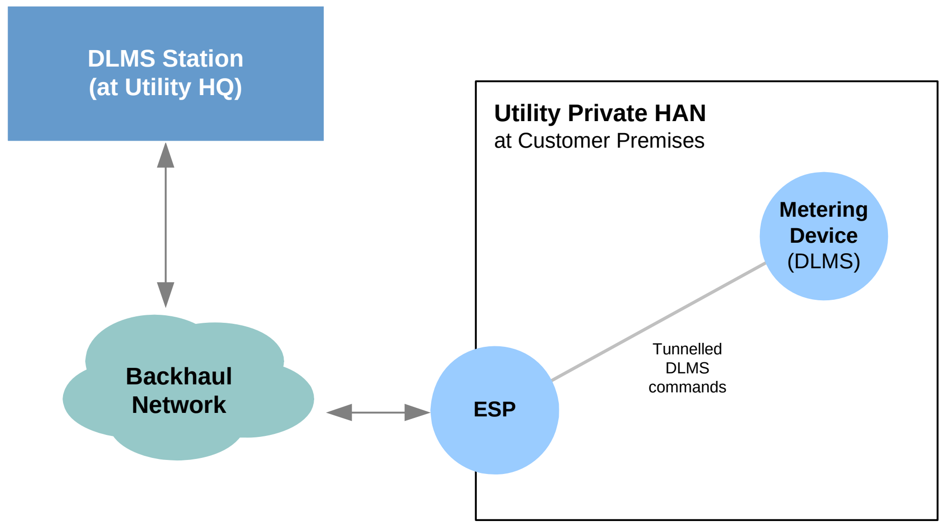

Figure 17: Example Network for DLMS Tunnelling

The Tunnelling cluster is an optional cluster for all SE devices, and any device can support this cluster as a client or as a server or both.

- When a device acts as a Tunnelling cluster client, the device requests a tunnel from a cluster server and requests its closure when it is no longer needed.
- When a device acts as a Tunnelling cluster server, the device provides a tunnel to a client on request and manages the tunnel.

In the above example, the ESP acts as a Tunnelling cluster client and the Metering Device acts as a Tunnelling cluster server.

The Tunnelling cluster is enabled by defining CLD_TUNNELING in the **zcl_options.h** file - see Section 3.5.1. Further compile-time options for the Tunnelling cluster are detailed in Section 11.12.

The SE API provides functions for implementing the cluster commands. These functions are referenced throughout this chapter and are detailed in Section 11.8.

11.2 Tunnelling Cluster Structure and Attribute

The Tunnelling cluster has only one server attribute, which is mandatory and is contained in the following `tsCLD_Tunnel` structure:

```
typedef struct
{
    uint16 u16CloseTunnelTimeout;
} tsCLD_Tunnel;
```

where `u16CloseTunnelTimeout` is the timeout period, in seconds, for an inactive tunnel - that is, it is the minimum time for which the server will allow an opened tunnel to remain unused before closing it and releasing the associated resources.

11.3 Initialisation

Provided that the Tunnelling cluster is enabled through the compile-time options (see Section 11.12), the cluster will be automatically initialised when the SE profile is initialised and the SE device is registered in the application - that is, by calling **eSE_Initialise()** and the relevant device registration function, for example:

- **eSE_RegisterEspMeterEndPoint()** on a combined ESP/Metering Device (cluster client)
- **eSE_RegisterMeterEndPoint()** on a Metering Device (cluster server)

As part of this initialisation, the Tunnelling cluster is created as a server or client or both.

11.4 Tunnel Creation

A tunnel is created by the Tunnelling cluster server but this creation must be initiated on a cluster client using the function **eSE_TunnelRequestTunnelSend().** Note that this is a client function and should be called only on a client, once key establishment has completed. The function call requires the following to be specified for the requested tunnel:

- The ID of the protocol used by the data which is to be tunnelled
- The maximum size, in bytes, of the incoming tunnelled data

The function call results in a tunnel creation request being sent to the server.

Server-side Events

Calling **eSE_TunnelRequestTunnelSend()** on the client then results in the generation of the following events on the server:

- E_SE_TUN_REQUEST_TUNNEL_REQUEST_RECEIVED

  This event indicates that the server has received a request to create a tunnel. The details of the tunnel request are given in the callback structure `tsSE_TunnelRequestTunnelCmdRcvd`. This event is passed to the application to check whether the Protocol ID requested by the client is supported on the server and to set the maximum incoming data size on the server. The application should update both these values in the fields `eStatus` and `u16MaxLocalIncmgDataSizeSupported` of the structure `tsSE_TunnelRequestTunnelCmdRcvd` before returning from the registered endpoint callback function.

- E_SE_TUN_CREATED

  This event indicates that the server has created a tunnel and gives the Tunnel ID created for the application. The application can then use this Tunnel ID during data transfer. Note that the application either can store this Tunnel ID for later data transfers or can retrieve the tunnel details when required using the function **eSE_TunnelGetInformation()**.

Client-side Event

Calling **eSE_TunnelRequestTunnelSend()** on the client also results in the generation of the following event on the client:

- E_SE_TUN_REQUEST_TUNNEL_RESPONSE_RECEIVED

  This event indicates the status of tunnel created. If the tunnel status is success, the client application can save the Tunnel ID for sending data or for closing the tunnel. The application can alternatively retrieve the tunnel information using the function **eSE_TunnelGetInformation()** when required (instead of storing it).

11.5 Tunnelled Data Transfer

A tunnelled data transfer can be initiated on either the client or the server after the successful creation of a tunnel (as described in Section 11.4) - that is, after the event E_SE_TUN_CREATED has been generated on the server or the event E_SE_TUN_REQUEST_TUNNEL_RESPONSE_RECEIVED has been generated on client. The data transfer can be initiated by calling the function **eSE_TunnelTransferDataSend()** on either device.

Note that the maximum data size that can be transmitted using **eSE_TunnelTransferDataSend()** should not exceed the maximum incoming data transfer size of the recipient and should not exceed

APDU size - 5 bytes

where 3 bytes are reserved for the ZCL header and 2 bytes are reserved for the Tunnel ID.

If the application needs to transfer more data than this, another call to **eSE_TunnelTransferDataSend()** to initiate a second data transfer may be needed after the event E_SE_TUN_TUNNEL_DATA_TRANSFER_COMPLETED or after a timeout which is sufficient for the previous data transfer to complete.

The function **eSE_TunnelTransferDataSend()** takes a pointer to the data. The data should be allocated by the application, which can use a global **uint8** array for storing the data.

Initiator Events

The following events are generated on the initiator of the tunnelled data transfer:

- E_SE_TUN_TUNNEL_DATA_TRANSFER_COMPLETED

  This event indicates that data transmission has completed. The application may choose to continue further data transmissions or close the tunnel.

- E_SE_TUN_TUNNEL_DATA_TRANSFER_ERROR

  This event indicates that the previous data transmission has resulted in an error on the recipient side. The status parameter indicates the type of error. The application may choose to retransmit the packet or ignore this event or close the tunnel.

Recipient Event

The following event is generated on the recipient of the tunnelled data transfer.

- E_SE_TUN_TRANSFER_DATA_COMMAND_RECEIVED

  This event indicates that tunnelled data has been received. The event contains the data length and a pointer to the received data. The application should save the data, if required, as the pointer will be deallocated once the registered endpoint callback function has completed.

  After receiving this event, the application may choose to send data to the initiator or receive more data (by waiting for another event of the type E_SE_TUN_TRANSFER_DATA_COMMAND_RECEIVED), or may choose to ignore the event or close the tunnel (if a client).

11.6 Closing a Tunnel

A client can request a tunnel to be closed by calling **eSE_TunnelCloseTunnelSend()**. Note that this is a client function and can be called only on a client. As a result of this function call, a Tunnel Close command is sent to the server.

The following event is generated on the server after receiving a Tunnel Close command:

- E_SE_TUN_CLOSED

  This event indicates that a tunnel has been closed on the server.

  Note that this event can be generated either:

  - on receiving a Tunnel Close command from a client, or
  - if the tunnel is closed on a timeout value (stored in the server attribute - see Section 11.2).

  The contents of E_SE_TUN_CLOSED give the reason for the tunnel closure (either E_SE_TUN_CLOSE_TUN_CMD_RECVD or E_SE_TUN_TIMEOUT).

Note: The application should not attempt to use the tunnel after the tunnel has been closed. If a tunnelled data transfer is required at a later point, the client application should then request a new tunnel using the function **eSE_TunnelRequestTunnelSend()**.

11.7 Tunnelling Events

The Tunnelling cluster has its own events that are handled through the callback mechanism outlined in Section 4.7 (and fully detailed in the *ZCL User Guide (JN-UG-3077)*). If a device uses the Tunnelling cluster then Tunnelling event handling must be included in the callback function for the associated endpoint - for example:

- For an Metering device (cluster server), this callback function is registered through **eSE_RegisterMeterEndPoint()**
- For an ESP/Meter (cluster client), this callback function is registered through **eSE_RegisterESPMeterEndPoint()**

The relevant callback function will then be invoked when a Tunnelling event occurs. For a Tunnelling event, the `eEventType` field of the `tsZCL_CallBackEvent` structure is set to E_ZCL_CBET_CLUSTER_CUSTOM. This event structure also contains an element `sClusterCustomMessage`, which is itself a structure containing a field `pvCustomData`. This field is a pointer to a `tsSE_TunnelCallBackMessage` structure which contains the Tunnelling parameters:

```
typedef struct
{
    teSE_TunnelCallbackEvents eEventType;
    union
    {
        tsSE_TunnelRequestTunnelCmdRcvd    sTunnelRequestTunnelCmd;
        tsSE_TunnelRequestTunnelResponse   sTunnelRequestTunnelRspCmd;
        tsSE_TunnelRequestTunnelCreated    sTunnelCreated;
        tsSE_TunnelTransferDataCmdPyldRcvd sTunnelTransferDataCmd;
        tsSE_TunnelTransferDataReqStatus   sTransferDataReq;
        tsSE_TunnelTransferDataError       sTunnelTransferDataErrCmd;
        tsSE_TunnelcloseTunnel             sCloseTunnelDetails;
    }uMessage;
}tsSE_TunnelCallBackMessage;
```

11.7.1 Events Types

The eEventType field of the above structure specifies the type of Tunnelling event that has been generated - these event types are enumerated in the teSE_TunnelCallbackEvents structure and are described below.

```
typedef enum PACK
{
    E_SE_TUN_REQUEST_TUNNEL_REQUEST_RECEIVED,
    E_SE_TUN_REQUEST_TUNNEL_RESPONSE_RECEIVED,
    E_SE_TUN_CREATED,
    E_SE_TUN_TRANSFER_DATA_COMMAND_RECEIVED,
    E_SE_TUN_TUNNEL_DATA_TRANSFER_COMPLETED,
    E_SE_TUN_TUNNEL_DATA_TRANSFER_ERROR,
    E_SE_TUN_CLOSED
}teSE_TunnelCallbackEvents;
```

E_SE_TUN_REQUEST_TUNNEL_REQUEST_RECEIVED

This event is generated on a server on receiving a Request Tunnel command. The event is passed to the application to check whether the application supports the protocol requested and to indicate the incoming data transfer size for the server. The event has the following payload:

```
typedef struct {
    teSE_TunnelStatus  eStatus;
    uint16             u16MaxLocalIncmgDataSizeSupported;
    tsSE_TunnelRequestTunnelCmdPyld sTunnelRequestTunnelCmdPyld;
}tsSE_TunnelRequestTunnelCmdRcvd;
```

where

- eStatus indicates whether the protocol is supported. The application should assign any of the following values to eStatus before returning from the callback function:

  E_SE_TUN_SUCCESS

  E_SE_TUN_BUSY

  E_SE_TUN_NO_MORE_TUNNEL

  E_SE_TUN_PROTOCOL_NOT_SUPPORTED
- u16MaxLocalIncmgDataSizeSupported is the maximum incoming data transfer size on the server. Note that this value should be less than (APDU size - 5)
- sTunnelRequestTunnelCmdPyld contains the details of a tunnel request received on the server. The tsSE_TunnelRequestTunnelCmdPyld structure is described in Section 11.10.2

E_SE_TUN_REQUEST_TUNNEL_RESPONSE_RECEIVED

This event is generated on a client and indicates the status of a Request Tunnel command that has been issued. The event has the following payload:

```
typedef struct
{
    teSE_TunnelRequestTunnelStatus  eStatus;
    uint16                          u16TunnelID;
    uint16                          u16MaxIncmgTransferSize;
}tsSE_TunnelRequestTunnelResponse;
```

where

- `eStatus` indicates the status of the Request Tunnel command. This can have any of the following enumerated values:

  E_SE_TUN_SUCCESS

  E_SE_TUN_BUSY

  E_SE_TUN_NO_MORE_TUNNEL

  E_SE_TUN_PROTOCOL_NOT_SUPPORTED

  E_SE_TUN_FLOW_CONTROL_NOT_SUPPORTED
- `u16TunnelID` is the identifier for the requested tunnel (but will only be valid if `eStatus` returns E_SE_TUN_SUCCESS)
- `u16MaxIncmgTransferSize` is the maximum incoming data transfer size on the server. The client should not initiate a data transfer with a data length greater than this value by means of a single Transfer Data command

E_SE_TUN_CREATED

This event is generated on a server and indicates that a tunnel has been created. The event has the following payload:

```
typedef struct
{
    uint16 u16TunnelID;
}tsSE_TunnelRequestTunnelCreated;
```

where `u16TunnelID` is the identifier assigned to the created tunnel.

E_SE_TUN_TRANSFER_DATA_COMMAND_RECEIVED

This event can be generated either on a server or on a client after tunnelled data has been received. The event has the following payload:

```
typedef struct
{
    uint8       u8Status;
    uint16      u16TunnelID;
    uint8      *pu8Data;
    uint16      u16DataLength;
    uint16      u16NumOfBytesLeft;
}tsSE_TunnelTransferDataCmdPyldRcvd;
```

where

- `u8Status` should be set by the application in the case of an error during the processing of the data
- `u16TunnelID` is the identifier of the tunnel through which the data was received
- `pu8Data` is a pointer to the received data. Note that this pointer will be invalid once the endpoint callback function has completed and so the data should be backed up by the application, if required at a later point
- `u16DataLength` is the length of the received data, in bytes
- `u16NumOfBytesLeft` is reserved for future use

E_SE_TUN_TUNNEL_DATA_TRANSFER_COMPLETED

This event is generated on the server or a client, whichever has initiated the data transfer using the function **eSE_TunnelTransferDataSend()**. The event has the following payload:

```
typedef struct
{
    uint8      u8Status;
    uint16     u16TunnelID;
}tsSE_TunnelTransferDataReqStatus;
```

where

- `u8Status` indicates the status of the transmission (this is the status returned by the ZPS_tsAfDataAckEvent event)
- `u16TunnelID` is the identifier of the tunnel used for the data transfer

E_SE_TUN_TUNNEL_DATA_TRANSFER_ERROR

This event can be generated either on a server or on a client after receiving a Data Transfer Error command. The event has the following payload:

```
typedef struct
{
    tsSE_TunnelTransferDataStatus    eTunnelTransferDataStatus;
    uint16                           u16TunnelID;
}tsSE_TunnelTransferDataError;
```

where

- `eTunnelTransferDataStatus` is the error status. This can have any of the following enumerated values:

  E_SE_TUN_NO_SUCH_TUNNEL

  E_SE_TUN_WRONG_DEVICE

  E_SE_TUN_DATA_ERR_OVERFLOW
- `u16TunnelID` is the identifier of the tunnel through which the Data Transfer Error command was received.

E_SE_TUN_CLOSED

This event is generated on the server on the closure of a tunnel, either due to the reception of a Close Tunnel command or due to a timeout (of value E_CLD_TUN_CLOSE_TUNNEL_TIMEOUT) on an inactive channel. The event has the following payload.

```
typedef struct
{
    uint16                    u16TunnelID;
    teSE_TunnelCloseCause     u8CloseReason;
}tsSE_TunnelcloseTunnel;
```

where

- `u16TunnelID` is the identifier of the tunnel that has been closed
- `u8CloseReason` indicates the reason for closure. This can be either E_SE_TUN_CLOSE_TUN_CMD_RECVD or E_SE_TUN_TIMEOUT

11.7.2 Example Event Handling Callback Function

```
PRIVATE void vHandleTunnelingEvent(void *pvParam)
{
tsSE_TunnelCallBackMessage *psMessage = (tsSE_TunnelCallBackMessage*) (pvParam);

tsSE_TunnelTransferDataReqCmdPyld sData;
uint8 u8SeqNo;
uint16 tunnelID;
switch( psMessage->eEventType)
{
case E_SE_TUN_REQUEST_TUNNEL_REQUEST_RECEIVED:


psMessage->uMessage.sTunnelRequestTunnelCmd.eStatus =
    E_SE_TUN_SUCCESS;
if(E_PROTOCOL_SUPPORTED == psMessage-
>uMessage.sTunnelRequestTunnelCmd.sTunnelRequestTunnelCmdPyld.u8ProtocolID)
{
    psMessage->uMessage.sTunnelRequestTunnelCmd.eStatus =
        E_SE_TUN_SUCCESS;
}
Else
{
    psMessage->uMessage.sTunnelRequestTunnelCmd.eStatus =
        E_SE_TUN_PROTOCOL_NOT_SUPPORTED;
}
psMessage->uMessage.sTunnelRequestTunnelCmd.u16MaxLocalIncmgDataSizeSupported  =
200;
break;
case E_SE_TUN_REQUEST_TUNNEL_RESPONSE_RECEIVED:

sData.u16TunnelID = psMessage->uMessage.sTunnelRequestTunnelRspCmd.u16TunnelID;
if(0x00 == psMessage->uMessage.sTunnelRequestTunnelRspCmd.eStatus)
{
                eSE_TunnelTransferDataSend(
                0x01,
                0x01,
                FALSE,
                &s_sDevice.sEsp.sAddress,
                &u8SeqNo,
                &sData);
}

case E_SE_TUN_CREATED:
DBG_vPrintf(TRUE,"E_SE_TUN_CREATED received\n");

DBG_vPrintf(TRUE,"u16TunnelID = %d\n",psMessage-
>uMessage.sTunnelCreated.u16TunnelID);
tunnelID = psMessage->uMessage.sTunnelCreated.u16TunnelID;
break;
case E_SE_TUN_TRANSFER_DATA_COMMAND_RECEIVED:

break;
case E_SE_TUN_TUNNEL_DATA_TRANSFER_COMPLETED:
eSE_TunnelCloseTunnelSend(
                           1,
                           1,
                           &s_sDevice.sEsp.sAddress,
                           &u8SeqNo,
                           sData.u16TunnelID);

break;
case E_SE_TUN_TUNNEL_DATA_TRANSFER_ERROR:

break;
case E_SE_TUN_GET_SUPPORTED_PROTOCOL_RESPONSE_RECEIVED:
break;
case E_SE_TUN_CLOSED:

break;

default:
break;
}
}
```

11.8 Functions

The following Tunnelling cluster functions are provided in the SE API and described in this section:

eSE_TunnelCreate

```
teZCL_Status eSE_TunnelCreate(
        bool_t bIsServer,
        uint8 *pu8AttributeControlBits,
        tsZCL_ClusterInstance *psTunnelClusterInstance,
        tsCLD_Tunnel *psTunnelData,
        tsSE_TunnelCustomDataStructure
                                  *psCustomDataStruct);
```

Description

This function creates an instance of the Tunnelling cluster on an endpoint. The cluster instance is created on the endpoint which is associated with the supplied `tsZCL_ClusterInstance` structure and can act as a server or a client, as specified.

Note that:

- The function is called internally by the SE device registration functions, and so an application that uses these standard functions does not need to explicitly call this function
- The function should only be explicitly called when the application has been registered with a custom endpoint, as described in Appendix B. In this case, it must be the first Tunnelling function called in the application, and must be called after the stack has been started and after the application profile has been initialised.

The function requires an array to be declared for internal use, which contains one element (of type **uint8**) for each attribute of the cluster. The array length should therefore equate to the total number of attributes supported by the Tunnelling cluster, which can be obtained by using the macro CLD_TUN_MAX_NUMBER_OF_ATTRIBUTE.

The array declaration should be as follows:

```
uint8
au8AppTunnel_ClusterAttributeControlBits[CLD_TUN_MAX_NUMBER_OF_ATTRIBUTE];
```

The function will initialise the array elements to zero.

Parameters

bIsServer	Type of cluster instance (server or client) to be created: TRUE - server FALSE - client
pu8AttributeControlBits	Pointer to an array of **uint8** values, with one element for each attribute in the cluster (see above).
psTunnelClusterInstance	Pointer to structure containing information about the cluster instance to be created (see the *ZCL User Guide (JN-UG-3077)*). This structure will be updated by the function by initialising individual structure fields.

psTunnelData	Pointer to attribute storage. This should be the address of a structure of type `tsCLD_Tunnel` which defines the attributes of the Tunnelling cluster. This function will initialise the attributes with default values.
psCustomDataStructure	Pointer to structure which contains custom data for the Tunnelling cluster. This structure is used for internal data storage. No knowledge of the fields of this structure is required.

Returns

E_ZCL_SUCCESS

E_ZCL_FAIL

E_ZCL_ERR_PARAMETER_NULL

E_ZCL_ERR_INVALID_VALUE

eSE_TunnelRequestTunnelSend

```
teSE_TunnelStatus eSE_TunnelRequestTunnelSend(
        uint8 u8SourceEndPointId,
        uint8 u8DestinationEndPointId,
        tsZCL_Address *psDestinationAddress,
        uint8 *pu8TransactionSequenceNumber,
        tsSE_TunnelRequestTunnelCmdPyld
                                    *sRequestTunnelCmdPyld);
```

Description

This function can be used by a client to request the creation of a tunnel by the server - that is, to send a Tunnel Request command to the server. The created tunnel will subsequently be available for communications between the server and client.

A pointer must be specified to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response will be set to match the TSN in the request, allowing an incoming response to be paired with a request.

Parameters

u8SourceEndPointId	Number of the local endpoint from which the Tunnel Request command is initiated
u8DestinationEndPointId	Number of the remote endpoint to which the Tunnel Request command will be sent
**psDestinationAddress*	Pointer to a structure containing the address of the remote node to which the Tunnel Request command will be sent
**pu8TransactionSequenceNumber*	Pointer to a location to store the Transaction Sequence Number (TSN) of the request
**sRequestTunnelCmdPyld*	Pointer to a structure containing data relating to the tunnel request (for details of the structure, see Section 11.10.2)

Returns

E_ZCL_SUCCESS
E_ZCL_FAIL
E_ZCL_ERR_PARAMETER_NULL
E_ZCL_ERR_CLUSTER_NOT_FOUND
E_SE_TUN_DEVICE_BUSY
E_SE_TUN_DATA_OVERFLOW
E_SE_TUN_INVALID_ADDRESS
E_SE_TUN_NOT_FOUND
E_SE_TUN_DATA_SUCCESS

eSE_TunnelTransferDataSend

```
teSE_TunnelStatus eSE_TunnelTransferDataSend(
            uint8 u8SourceEndPointId,
            uint8 u8DestinationEndPointId,
            bool bIsServer,
            tsZCL_Address *psDestinationAddress,
            uint8 *pu8TransactionSequenceNumber,
            tsSE_TunnelTransferDataReqCmdPyld
                                    *psTunnelCommandPyld);
```

Description

This function can be called on a server or client to send tunnelled data to the other node (a tunnel must have already been created between the two nodes).

A pointer must be specified to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response will be set to match the TSN in the request, allowing an incoming response to be paired with a request.

Parameters

u8SourceEndPointId	Number of the local endpoint from which the tunnelled data transfer will take place
u8DestinationEndPointId	Number of the remote endpoint to which the tunnelled data will be sent
bIsServer	Type of cluster instance (server or client) which is initiating the data transfer: TRUE - server FALSE - client
psDestinationAddress	Pointer to a structure containing the address of the remote node to which the tunnelled data will be sent
pu8TransactionSequenceNumber	Pointer to a location to store the Transaction Sequence Number (TSN) of the request
psTunnelCommandPyld	Pointer to a structure containing the data to be transferred (for details of the structure, see Section 11.10.3)

Returns

E_ZCL_SUCCESS
E_ZCL_FAIL
E_ZCL_ERR_PARAMETER_NULL
E_ZCL_ERR_CLUSTER_NOT_FOUND
E_SE_TUN_DEVICE_BUSY
E_SE_TUN_DATA_OVERFLOW
E_SE_TUN_INVALID_ADDRESS
E_SE_TUN_NOT_FOUND
E_SE_TUN_DATA_SUCCESS

eSE_TunnelCloseTunnelSend

```
teSE_TunnelStatus eSE_TunnelCloseTunnelSend(
                uint8 u8SourceEndPointId,
                uint8 u8DestinationEndPointId,
                tsZCL_Address *psDestinationAddress,
                uint8 *pu8TransactionSequenceNumber,
                uint16 u16TunnelId);
```

Description

This function can be used by a client to request a tunnel to be closed by the server.

Parameters

u8SourceEndPointId	Number of the local endpoint from which the tunnel will be closed
u8DestinationEndPointId	Number of the remote endpoint to which the tunnel is connected
psDestinationAddress	Pointer to a structure containing the address of the remote node to which the tunnel is connected
pu8TransactionSequenceNumber	Pointer to a location to store the Transaction Sequence Number (TSN) of the request
u16TunnelId	The identifier of the tunnel to be closed

Returns

E_ZCL_SUCCESS

E_ZCL_FAIL

E_ZCL_ERR_PARAMETER_NULL

E_ZCL_ERR_CLUSTER_NOT_FOUND

E_SE_TUN_DEVICE_BUSY

E_SE_TUN_NOT_FOUND

eSE_TunnelGetInformation

```
teSE_TunnelStatus eSE_TunnelGetInformation(
                        uint8 u8SourceEndPointId,
                        uint8 u8DestinationEndPointId,
                        bool bIsServer,
                        tsZCL_Address *psDestinationAddress,
                        uint16 u16TunnelId,
                        tsSE_TunnelDetails *psTunnelDetails);
```

Description

This function can be used on a server or a client to obtain information about the specified tunnel which was created by the specified remote node.

The search for information can be also be used in the following ways:

- If the address of the remote node is set to NULL, the function will retrieve the first tunnel record matching the specified tunnel ID
- If the tunnel ID is set to 0xFFFF, the function will retrieve the first tunnel record with the specified remote node address
- If the address of the remote node is set to NULL and tunnel ID is set to 0xFFFF, the function will retrieve the first tunnel record

If no matching record is found, the function returns E_SE_TUN_NOT_FOUND.

Parameters

u8SourceEndPointId	Number of the local endpoint to which the tunnel is connected
u8DestinationEndPointId	Number of the remote endpoint to which the tunnel is connected
bIsServer	Type of cluster instance (server or client) which is initiating the information request: TRUE - server FALSE - client
psDestinationAddress	Pointer to a structure containing the address of the remote node to which the tunnel is connected
pu8TransactionSequenceNumber	Pointer to a location to store the Transaction Sequence Number (TSN) of the request
u16TunnelId	The identifier of the tunnel for which information is requested
psTunnelDetails	Pointer to structure in which the obtained details of the tunnel will be stored

Returns

E_ZCL_SUCCESS

E_SE_TUN_NOT_FOUND

11.9 Return Codes

In addition to some of the ZCL status enumerations (detailed in the *ZCL User Guide (JN-UG-3077)*), the following enumerations are returned by the SE API Tunnelling cluster functions (see Section 11.8) to indicate the outcome of the function call.

```
typedef enum PACK
{
    E_SE_TUN_DEVICE_BUSY =  E_ZCL_ERR_ENUM_END + 1,
    E_SE_TUN_DATA_OVERFLOW,
    E_SE_TUN_INVALID_ADDRESS,
    E_SE_TUN_NOT_FOUND,
    E_SE_TUN_DATA_SUCCESS
}teSE_TunnelStatus;
```

The above enumerations are described in the table below.

Enumeration	Description
E_SE_TUN_DEVICE_BUSY	Device is busy
E_SE_TUN_DATA_OVERFLOW	Data size is greater than maximum incoming data size which was negotiated when the tunnel was created
E_SE_TUN_INVALID_ADDRESS	Specified address is not valid
E_SE_TUN_NOT_FOUND	Tunnel does not exist
E_SE_TUN_DATA_SUCCESS	Tunnelled data transfer was a success

Table 46: Tunnelling Cluster Return Codes

11.10 Structures

11.10.1 tsSE_TunnelCallBackMessage

This structure is used when a tunnelling event is generated - for example, as the result of a tunnelling command having been received.

```
typedef struct
{
    teSE_TunnelCallbackEvents eEventType;
    union
    {
        tsSE_TunnelRequestTunnelCmdRcvd      sTunnelRequestTunnelCmd;
        tsSE_TunnelRequestTunnelResponse     sTunnelRequestTunnelRspCmd;
        tsSE_TunnelRequestTunnelCreated      sTunnelCreated;
        tsSE_TunnelTransferDataCmdPyldRcvd   sTunnelTransferDataCmd;
        tsSE_TunnelTransferDataReqStatus     sTransferDataReq;
        tsSE_TunnelTransferDataError         sTunnelTransferDataErrCmd;
        tsSE_TunnelcloseTunnel               sCloseTunnelDetails;
    }uMessage;
}tsSE_TunnelCallBackMessage;
```

Where:

- `eEventType` is the Tunnelling event type - see Section 11.11.1.
- `uMessage` is a union containing the event details in one of the following forms (depending on the event specified in the field `eEventType`):
  - `sTunnelRequestTunnelCmd` is a structure containing the payload of a received Request Tunnel command
  - `sTunnelRequestTunnelRspCmd` is a structure containing the payload of the response to a Request Tunnel command (response received by client)
  - `sTunnelCreated` is a structure containing the details of a tunnel created by the server
  - `sTunnelTransferDataCmd` is a structure containing the details of received tunnelled data
  - `sTunnelTransferDataErrCmd` is a structure containing the details of a received tunnel data error command
  - `sCloseTunnelDetails` is a structure containing the details of a tunnel closed by the server

11.10.2 tsSE_TunnelRequestTunnelCmdPyld

This structure is used to hold the client's requirements when the client requests a tunnel to be created.

```
typedef struct {
    uint8     u8ProtocolID;
    uint16    u16ManufCode;
    bool_t    bFlowControlSupport;
    uint16    u16MaxIncmgTransferSize;
}tsSE_TunnelRequestTunnelCmdPyld;
```

where

- `u8ProtocolID` is an enumeration representing the identifier of the metering communication protocol for which the tunnel is requested - see Section 11.11.5.
- `u16ManufCode` is a manufacture code which is related either to the manufacturer of a device and/or to a manufacturer-specific protocol. The value 0xFFFF is used to indicate that a manufacture code is not used.
- `bFlowControlSupport` indicates whether flow control support is requested from the tunnel. *In this release, flow control is not supported and this field should be set to FALSE.*
- `u16MaxIncmgTransferSize` is the maximum size of a data packet, in bytes, that can be transferred to the client in the payload of a single Transfer Data command.

11.10.3 tsSE_TunnelTransferDataReqCmdPyld

This structure is used to send a Transfer Data command and holds the details of the data to be tunnelled.

```
typedef struct
{
    uint16    u16TunnelID;
    uint8     *pu8Data;
    uint16    u16dataLength;
}tsSE_TunnelTransferDataReqCmdPyld;
```

where:

- `u16TunnelID` is the identifier of the tunnel to be used (created when the tunnel was created).
- `pu8Data` is a pointer to the data to be sent
- `u16dataLength` is the length of the data to be sent, in bytes

11.10.4 tsSE_TunnelRequestTunnelResponse

This structure is used on a client on generating the event E_SE_TUN_REQUEST_TUNNEL_RESPONSE_RECEIVED, containing the details of the response to a Request Tunnel command.

```
typedef struct
{
    teSE_TunnelRequestTunnelStatus    eStatus;
    uint16                            u16TunnelID;
    uint16                            u16MaxIncmgTransferSize;
}tsSE_TunnelRequestTunnelResponse
```

where:

- `eStatus` is the status of the Request Tunnel command. Enumerations are provided and are detailed in Section 11.11.2.
- `u16TunnelID` is the identifier for the tunnel.
- `u16MaxIncmgTransferSize` is the maximum size of a data packet, in bytes, that can be transferred to the server in the payload of a single Transfer Data command.

11.10.5 tsSE_TunnelRequestTunnelCreated

This structure is used on generating the event E_SE_TUN_CREATED, containing the details of details of a created tunnel.

```
typedef struct
{
    uint16 u16TunnelID;
}tsSE_TunnelRequestTunnelCreated;
```

where `u16TunnelID` is the identifier of the created tunnel.

11.10.6 tsSE_TunnelTransferDataCmdPyldRcvd

This structure is used on generating the event E_SE_TUN_TRANSFER_DATA_COMMAND_RECEIVED, containing received tunnelled data.

```
typedef struct
{
    uint8     u8Status;
    uint16    u16TunnelID;
    uint8     *pu8Data;
    uint16    u16DataLength;
    uint16    u16NumOfBytesLeft;
}tsSE_TunnelTransferDataCmdPyldRcvd;
```

where:

- `u8Status` is the status of the tunnelled data transfer.
- `u16TunnelID` is the identifier of the tunnel through which the data was received.
- `pu8Data` is a pointer to received data.
- `u16DataLength` is the length of the received data, in bytes.
- `u16NumOfBytesLeft` is the number of data bytes remaining to be transferred (and requiring another Transfer Data command).

11.10.7 tsSE_TunnelTransferDataReqStatus

This structure is used on generating the event E_SE_TUN_TUNNEL_DATA_TRANSFER_COMPLETED, containing the status of a tunnelled data transmission.

```
typedef struct
{
    uint8     u8Status;
    uint16    u16TunnelID;
}tsSE_TunnelTransferDataReqStatus;
```

where

- `u8Status` is the status of the data transmission (it will be the status returned by the stack ZPS_tsAfDataAckEvent event).
- `u16TunnelID` is the identifier of the tunnel ID through which the Transfer Data command was initiated.

11.10.8 tsSE_TunnelTransferDataError

This structure is used on generating the event E_SE_TUN_TUNNEL_DATA_TRANSFER_ERROR, containing the details of a received Transfer Data Error command.

```
typedef struct
{
    tsSE_TunnelTransferDataStatus     eTunnelTransferDataStatus;
    uint16                            u16TunnelID;
}tsSE_TunnelTransferDataError
```

where

- `eTunnelTransferDataStatus` is the status code received in the Transfer Data Error command. Enumerations are provided and are detailed in Section 11.11.3.
- `u16TunnelID` is the identifier of the tunnel through which the Transfer Data Error command was received.

11.10.9 tsSE_TunnelcloseTunnel

This structure is used on a server on generating the event E_SE_TUN_CLOSED, containing details of a tunnel that has been closed.

```
typedef struct
{
    uint16                     u16TunnelID;
    teSE_TunnelCloseCause      eCloseReason;
}tsSE_TunnelcloseTunnel;
```

where:

- `u16TunnelID` is the identifier of the tunnel which has been closed
- `eCloseReason` indicates the reason for closing the tunnel. Enumerations are provided and are detailed in Section 11.11.4.

11.10.10 tsSE_TunnelDetails

This structure contains the details of the tunnel record returned when the function **eSE_TunnelGetInformation()** is called.

```
typedef struct
{
    teSE_RemoteTunnelStatus eStatus;
    uint16                  u16TunnelID;
    uint64                  u64DeviceAddress;
    uint8                   u8EndPoint;
    uint8                   bFlowControl;
    uint32                  u32UtcTime;
    uint16                  u16MaxIncmgTransferSizeRemoteDevice;
    uint16                  u16MaxIncmgTransferSize;
    uint8                   u8SeqNo;
}tsSE_TunnelDetails;
```

where

- `u16TunnelID` is the identifier of the tunnel
- `u64DeviceAddress` is the 64-bit IEEE/MAC address of the device that created the tunnel
- `u8EndPoint` is the number of the endpoint on which the tunnel was created (on the device that created the tunnel).
- `u16MaxIncmgTransferSizeRemoteDevice` is the maximum size, in bytes, of an incoming data transfer for the remote device
- `u16MaxIncmgTransferSize` is the maximum size, in bytes, of an incoming data transfer for the local device.
- All other fields are used for internal cluster management and can be ignored by the application.

11.11 Enumerations

11.11.1 'Tunnelling Event' Enumerations

The event types generated by the Tunnelling cluster are enumerated in the `teSE_TunnelCallbackEvents` structure below:

```
typedef enum PACK
{
    E_SE_TUN_REQUEST_TUNNEL_REQUEST_RECEIVED,
    E_SE_TUN_REQUEST_TUNNEL_RESPONSE_RECEIVED,
    E_SE_TUN_CREATED,
    E_SE_TUN_TRANSFER_DATA_COMMAND_RECEIVED,
    E_SE_TUN_TUNNEL_DATA_TRANSFER_COMPLETED,
    E_SE_TUN_TUNNEL_DATA_TRANSFER_ERROR,
    E_SE_TUN_CLOSED
}teSE_TunnelCallbackEvents;
```

The above event types are described in the table below.

Enumeration	**Description**
E_SE_TUN_REQUEST_TUNNEL_REQUEST_RECEIVED	Generated on server on receiving a Request Tunnel command
E_SE_TUN_REQUEST_TUNNEL_RESPONSE_RECEIVED	Generated on client and indicates the status of a Request Tunnel command previously issued
E_SE_TUN_CREATED	Generated on server and indicates that a tunnel has been created
E_SE_TUN_TRANSFER_DATA_COMMAND_RECEIVED	Generated either on server or on client after a tunnelled data transfer is received.
E_SE_TUN_TUNNEL_DATA_TRANSFER_COMPLETED	Generated on server or client after the completion of a data transfer initiated using the function **eSE_TunnelTransferDataSend()**
E_SE_TUN_TUNNEL_DATA_TRANSFER_ERROR	Generated when a Data Transfer Error command has been received
E_SE_TUN_CLOSED	Generated on server on the closure of a tunnel either due to the reception of a Close Tunnel command or due to a timeout on an inactive channel (E_CLD_TUN_CLOSE_TUNNEL_TIMEOUT)

Table 47: Tunnelling Event Enumerations

11.11.2 'Request Tunnel Status' Enumerations

The status values of the response to a Request Tunnel command are enumerated in the `teSE_TunnelRequestTunnelStatus` structure below:

```
typedef enum PACK
{
    E_SE_TUN_SUCCESS = 0x00,
    E_SE_TUN_BUSY,
    E_SE_TUN_NO_MORE_TUNNEL,
    E_SE_TUN_PROTOCOL_NOT_SUPPORTED,
    E_SE_TUN_FLOW_CONTROL_NOT_SUPPORTED
}teSE_TunnelRequestTunnelStatus;
```

The above enumerations are listed and described in the table below.

Enumeration	Description
E_SE_TUN_SUCCESS	Tunnel created successfully
E_SE_TUN_BUSY	Tunnel is busy - try again after a delay
E_SE_TUN_NO_MORE_TUNNEL	All possible tunnels are in use
E_SE_TUN_PROTOCOL_NOT_SUPPORTED	Requested protocol is not supported by server
E_SE_TUN_FLOW_CONTROL_NOT_SUPPORTED	Flow control is not supported by server

Table 48: 'Request Tunnel Status' Enumerations

11.11.3 'Data Transfer Error' Enumerations

The status codes of the Data Transfer Error command are enumerated in the `tsSE_TunnelTransferDataStatus` structure below.

```
typedef enum PACK
{
    E_SE_TUN_NO_SUCH_TUNNEL,
    E_SE_TUN_WRONG_DEVICE,
    E_SE_TUN_DATA_ERR_OVERFLOW
}tsSE_TunnelTransferDataStatus
```

The above enumerations are listed and described in the table below.

Enumeration	Description
E_SE_TUN_NO_SUCH_TUNNEL	Data received through invalid tunnel
E_SE_TUN_WRONG_DEVICE	Data received through a tunnel which have no access to
E_SE_TUN_DATA_ERR_OVERFLOW	Received data length is greater than maximum permissible incoming data transfer size

Table 49: 'Data Transfer Error' Enumerations

11.11.4 'Close Cause' Enumerations

The reasons for a tunnel closure by a server are enumerated in the `teSE_TunnelCloseCause` structure below:

```
typedef enum PACK
{
    E_SE_TUN_CLOSE_TUN_CMD_RECVD,
    E_SE_TUN_TIMEOUT
}teSE_TunnelCloseCause;
```

The above enumerations are listed and described in the table below.

Enumeration	Description
E_SE_TUN_CLOSE_TUN_CMD_RECVD	Close Tunnel command received from client
E_SE_TUN_TIMEOUT	Tunnel is closed due to inactivity (timeout)

Table 50: 'Close Cause' Enumerations

11.11.5 'Protocol ID' Enumerations

The protocols defined by ZigBee for tunnelling are enumerated in the `teSE_tunnelProtocolID` structure below:

```
typedef enum PACK
{
    E_CLD_TUN_DLMS_COSEM_IEC_62056 = 0,
    E_CLD_TUN_IEC_61107,                        // 1
    E_CLD_TUN_ANS_C12,                          // 2
    E_CLD_TUN_M_BUS,                            // 3
    E_CLD_TUN_SML,                              // 4
    E_CLD_TUN_CLIMATE_TALK,                     // 5
    E_CLD_TUN_MANUFACTURER_DEFINED_BASE = 200
} teSE_tunnelProtocolID;
```

The above enumerations are listed and described in the table below.

Enumeration	Protocol
E_CLD_TUN_DLMS_COSEM_IEC_62056	DLMS/COSEM (IEC 62056)
E_CLD_TUN_IEC_61107	IEC 61107
E_CLD_TUN_ANS_C12	ANSI C12
E_CLD_TUN_M_BUS	M-BUS
E_CLD_TUN_SML	SML
E_CLD_TUN_CLIMATE_TALK	ClimateTalk
E_CLD_TUN_MANUFACTURER_DEFINED_BASE	Manufacturer-defined protocols

Table 51: 'Protocol' Enumerations

11.12 Compile-Time Options

This section describes the compile-time options that may be set in the **zcl_options.h** file of an application that uses the Tunnelling cluster.

The Tunnelling cluster is enabled by defining CLD_TUNNELING.

Client and server versions of the cluster are defined using TUNNELING_CLIENT and TUNNELING_SERVER respectively.

Maximum Simultaneous Tunnels

The maximum number of tunnels that can exist at the same time on a device can be defined using the macro:

CLD_TUN_MAX_SIMULTANEOUS_TUNNELS

If the application does not define this macro, the default value of one is used.

Part III: General Reference Information

12. Initialisation and Device Registration Functions

This chapter details the core functions of the Smart Energy API. These comprise the following initialisation function and device-specific endpoint registration functions:

Note 1: For guidance on using these functions in your application code, refer to Chapter 4.

Note 2: The return codes for these functions are described in the *ZCL User Guide (JN-UG-3077)*.

Note 3: The SE API provides a separate endpoint registration function for each SE device type. Functions are provided for a combined ESP/Metering Device and for separate ESP and Metering Device.

Note 4: SE initialisation must also be performed through definitions in the header file **zcl_options.h** - see Section 3.5.1. In addition, JenOS resources for SE (such as a software timer for a real-time clock) must also be pre-configured using the JenOS Configuration Editor - refer to the *JenOS User Guide (JN-UG-3075)*.

eSE_Initialise

```
teZCL_Status eSE_Initialise(
                    tfpZCL_ZPSCallBackFunction cbCallBack,
                    PDUM_thAPdu hAPdu);
```

Description

This function initialises the ZCL and SE libraries. It should be called before registering any endpoints (using one of the device-specific endpoint registration functions from this section).

As part of this function call, you must specify a user-defined callback function that will be invoked when a ZigBee PRO stack event occurs that is not associated with an endpoint (the callback function for events associated with an endpoint is specified when the endpoint is registered using one of the registration functions). This callback function is defined according to the typedef:

```
typedef void (* tfpZCL_ZCLCallBackFunction)
                (tsZCL_CallBackEvent *pCallBackEvent);
```

You must also provide a pointer to a local pool of Application Protocol Data Units (APDUs) that will be used by the ZCL to hold messages to be sent and received.

Parameters

cbCallBack	Pointer to a callback function to handle stack events that are not associated with a registered endpoint
hAPdu	Pointer to a pool of APDUs for holding messages to be sent and received

Returns

E_ZCL_SUCCESS

E_ZCL_ERR_HEAP_FAIL

E_ZCL_ERR_PARAMETER_NULL

eSE_RegisterEspMeterEndPoint

```
teZCL_Status eSE_RegisterEspMeterEndPoint(
                    uint8 u8EndPointIdentifier,
                    tfpZCL_ZCLCallBackFunction cbCallBack,
                    tsSE_EspMeterDevice *psDeviceInfo,
                    uint8 u8MirrorStartEndPoint);
```

Description

This function is used to register an endpoint which will support a combined ESP/ Metering Device. The function must be called after the **eSE_Initialise()** function and before starting the ZigBee PRO stack.

The specified identifier for the endpoint is a number in the range 1 to 240 (endpoint 0 is reserved for ZigBee use). SE endpoints are normally numbered consecutively starting at 1. The specified number must be less than or equal to the value of SE_NUMBER_OF_ENDPOINTS defined in the **zcl_options.h** file, which represents the highest endpoint number used for SE.

As part of this function call, you must specify a user-defined callback function that will be invoked when an event occurs that is associated with the endpoint (events are detailed in the *ZCL User Guide (JN-UG-3077)*). This callback function is defined according to the typedef:

```
typedef void (* tfpZCL_ZCLCallBackFunction)
              (tsZCL_CallBackEvent *pCallBackEvent);
```

You must also provide a pointer to a `tsSE_EspMeterDevice` structure (see Section 13.2.1) which will be used to store all variables relating to the ESP and Metering Device associated with the endpoint. The `sEndPoint` and `sClusterInstance` fields of this structure are set by this function and must not be directly written to by the application.

If the ESP is to implement the mirroring of metering data for sleepy Metering devices, the number of the first endpoint to be used for mirroring must be specified. Depending on the maximum number of mirrors set in the compile-time options (see Section 8.12), consecutive endpoints will be reserved for mirrors. For example, if 5 is specified as the first mirror endpoint and up to 4 mirrors can be used then endpoints 5, 6, 7 and 8 will be reserved for mirrors. Mirroring is described in Section 8.5.

The function may be called multiple times if more than one endpoint is being used - for example, if more than one Metering Device is housed in the same hardware, sharing the same JN51xx module.

Parameters

u8EndPointIdentifier	Identifier of endpoint to be registered - this is an endpoint number in the range 1 to 240
cbCallBack	Pointer to a callback function to handle events associated with the registered endpoint
psDeviceInfo	Pointer to structure to be used to hold ESP and Metering Device variables

u8MirrorStartEndPoint	Number of first endpoint to be used for mirroring (to disable mirroring, set to 0)

Returns

E_ZCL_SUCCESS

E_ZCL_FAIL

E_ZCL_ERR_PARAMETER_NULL

E_ZCL_ERR_PARAMETER_RANGE

E_ZCL_ERR_EP_RANGE

E_ZCL_ERR_CLUSTER_0

E_ZCL_ERR_CALLBACK_NULL

eSE_RegisterEspEndPoint

```
teZCL_Status eSE_RegisterEspEndPoint(
                        uint8 u8EndPointIdentifier,
                        tfpZCL_ZCLCallBackFunction cbCallBack,
                        tsSE_EspDevice *psDeviceInfo,
                        uint8 u8MirrorStartEndPoint);
```

Description

This function is used to register an endpoint which will support a standalone ESP. The function must be called after the **eSE_Initialise()** function and before starting the ZigBee PRO stack.

The specified identifier for the endpoint is a number in the range 1 to 240 (endpoint 0 is reserved for ZigBee use). SE endpoints are normally numbered consecutively starting at 1. The specified number must be less than or equal to the value of SE_NUMBER_OF_ENDPOINTS defined in the **zcl_options.h** file, which represents the highest endpoint number used for SE.

As part of this function call, you must specify a user-defined callback function that will be invoked when an event occurs that is associated with the endpoint (events are detailed in the *ZCL User Guide (JN-UG-3077)*). This callback function is defined according to the typedef:

```
typedef void (* tfpZCL_ZCLCallBackFunction)
              (tsZCL_CallBackEvent *pCallBackEvent);
```

You must also provide a pointer to a `tsSE_EspDevice` structure (see Section 13.2.2) which will be used to store all variables relating to the ESP associated with the endpoint. The `sEndPoint` and `sClusterInstance` fields of this structure are set by this function and must not be directly written to by the application.

If the ESP is to implement the mirroring of metering data for sleepy Metering devices, the number of the first endpoint to be used for mirroring must be specified. Depending on the maximum number of mirrors set in the compile-time options (see Section 8.12), consecutive endpoints will be reserved for mirrors. For example, if 5 is specified as the first mirror endpoint and up to 4 mirrors can be used then endpoints 5, 6, 7 and 8 will be reserved for mirrors. Mirroring is described in Section 8.5.

The function may be called multiple times if more than one endpoint is being used - for example, if more than one ESP is housed in the same hardware, sharing the same JN51xx module.

Parameters

u8EndPointIdentifier	Identifier of endpoint to be registered - this is an endpoint number in the range 1 to 240
cbCallBack	Pointer to a callback function to handle events associated with the registered endpoint
psDeviceInfo	Pointer to structure to be used to hold ESP variables
u8MirrorStartEndPoint	Number of first endpoint to be used for mirroring (to disable mirroring, set to 0)

Returns

E_ZCL_SUCCESS

E_ZCL_FAIL

E_ZCL_ERR_PARAMETER_NULL

E_ZCL_ERR_PARAMETER_RANGE

E_ZCL_ERR_EP_RANGE

E_ZCL_ERR_CLUSTER_0

E_ZCL_ERR_CALLBACK_NULL

eSE_RegisterMeterEndPoint

```
teZCL_Status eSE_RegisterMeterEndPoint(
                    uint8 u8EndPointIdentifier,
                    tfpZCL_ZCLCallBackFunction cbCallBack,
                    tsSE_MeterDevice *psDeviceInfo);
```

Description

This function is used to register an endpoint which will support a standalone Metering Device. The function must be called after the **eSE_Initialise()** function and before starting the ZigBee PRO stack.

The specified identifier for the endpoint is a number in the range 1 to 240 (endpoint 0 is reserved for ZigBee use). SE endpoints are normally numbered consecutively starting at 1. The specified number must be less than or equal to the value of SE_NUMBER_OF_ENDPOINTS defined in the **zcl_options.h** file, which represents the highest endpoint number used for SE.

As part of this function call, you must specify a user-defined callback function that will be invoked when an event occurs that is associated with the endpoint (events are detailed in the *ZCL User Guide (JN-UG-3077)*). This callback function is defined according to the typedef:

```
typedef void (* tfpZCL_ZCLCallBackFunction)
               (tsZCL_CallBackEvent *pCallBackEvent);
```

You must also provide a pointer to a `tsSE_MeterDevice` structure (see Section 13.2.3) which will be used to store all variables relating to the Metering Device associated with the endpoint. The `sEndPoint` and `sClusterInstance` fields of this structure are set by this function and must not be directly written to by the application.

The function may be called multiple times if more than one endpoint is being used - for example, if more than one Metering Device is housed in the same hardware, sharing the same JN51xx module.

Parameters

u8EndPointIdentifier	Identifier of endpoint to be registered - this is an endpoint number in the range 1 to 240
cbCallBack	Pointer to a callback function to handle events associated with the registered endpoint
psDeviceInfo	Pointer to structure to be used to hold Metering Device variables

Returns

E_ZCL_SUCCESS
E_ZCL_FAIL
E_ZCL_ERR_PARAMETER_NULL
E_ZCL_ERR_PARAMETER_RANGE
E_ZCL_ERR_EP_RANGE
E_ZCL_ERR_CLUSTER_0
E_ZCL_ERR_CALLBACK_NULL

eSE_RegisterIPDEndPoint

```
teZCL_Status eSE_RegisterIPDEndPoint(
                    uint8 u8EndPointIdentifier,
                    tfpZCL_ZCLCallBackFunction cbCallBack,
                    tsSE_IPDDevice *psDeviceInfo);
```

Description

This function is used to register an endpoint which will support an IPD. The function must be called after the **eSE_Initialise()** function and before starting the ZigBee PRO stack.

The specified identifier for the endpoint is a number in the range 1 to 240 (endpoint 0 is reserved for ZigBee use). SE endpoints are normally numbered consecutively starting at 1. The specified number must be less than or equal to the value of SE_NUMBER_OF_ENDPOINTS defined in the **zcl_options.h** file, which represents the highest endpoint number used for SE.

As part of this function call, you must specify a user-defined callback function that will be invoked when an event occurs that is associated with the endpoint (events are detailed in the *ZCL User Guide (JN-UG-3077)*). This callback function is defined according to the typedef:

```
typedef void (* tfpZCL_ZCLCallBackFunction)
               (tsZCL_CallBackEvent *pCallBackEvent);
```

You must also provide a pointer to a `tsSE_IPDDevice` structure (see Section 13.2.4) which will be used to store all variables relating to the IPD associated with the endpoint. The `sEndPoint` and `sClusterInstance` fields of this structure are set by this function and must not be directly written to by the application.

The function may be called multiple times if more than one endpoint is being used.

Parameters

u8EndPointIdentifier	Identifier of endpoint to be registered - this is an endpoint number in the range 1 to 240
cbCallBack	Pointer to a callback function to handle events associated with the registered endpoint
psDeviceInfo	Pointer to structure to be used to hold IPD variables

Returns

E_ZCL_SUCCESS

E_ZCL_FAIL

E_ZCL_ERR_PARAMETER_NULL

E_ZCL_ERR_PARAMETER_RANGE

E_ZCL_ERR_EP_RANGE

E_ZCL_ERR_CLUSTER_0

E_ZCL_ERR_CALLBACK_NULL

eSE_RegisterRangeExtEndPoint

```
teZCL_Status eSE_RegisterRangeExtEndPoint(
                        uint8 u8EndPointIdentifier,
                        tfpZCL_ZCLCallBackFunction cbCallBack,
                        tsSE_EspMeterDevice *psDeviceInfo);
```

Description

This function is used to register an endpoint which will support a Range Extender. The function must be called after the **eSE_Initialise()** function and before starting the ZigBee PRO stack.

The specified identifier for the endpoint is a number in the range 1 to 240 (endpoint 0 is reserved for ZigBee use). SE endpoints are normally numbered consecutively starting at 1. The specified number must be less than or equal to the value of SE_NUMBER_OF_ENDPOINTS defined in the **zcl_options.h** file, which represents the highest endpoint number used for SE.

As part of this function call, you must specify a user-defined callback function that will be invoked when an event occurs that is associated with the endpoint (events are detailed in the *ZCL User Guide (JN-UG-3077)*). This callback function is defined according to the typedef:

```
typedef void (* tfpZCL_ZCLCallBackFunction)
              (tsZCL_CallBackEvent *pCallBackEvent);
```

You must also provide a pointer to a `tsSE_RangeExtDevice` structure (see Section 13.2.5) which will be used to store all variables relating to the Range Extender associated with the endpoint. The `sEndPoint` and `sClusterInstance` fields of this structure are set by this function and must not be directly written to by the application.

The function may be called multiple times if more than one endpoint is being used.

Parameters

u8EndPointIdentifier	Identifier of endpoint to be registered - this is an endpoint number in the range 1 to 240
cbCallBack	Pointer to a callback function to handle events associated with the registered endpoint
psDeviceInfo	Pointer to structure to be used to hold Range Extender variables

Returns

E_ZCL_SUCCESS
E_ZCL_FAIL
E_ZCL_ERR_PARAMETER_NULL
E_ZCL_ERR_PARAMETER_RANGE
E_ZCL_ERR_EP_RANGE
E_ZCL_ERR_CLUSTER_0
E_ZCL_ERR_CALLBACK_NULL

13. Structures, Enumerations and Parameters

This chapter details the SE API structures that are not specific to any particular SE cluster.

Note: Cluster-specific structures are detailed in the chapters for the respective clusters.

13.1 ZCL Structures

The following structures are defined in the ZigBee Cluster Library and are detailed in the *ZCL User Guide (JN-UG-3077)*:

- tsZCL_Address
- tsZCL_IndividualAttributesResponse
- tsZCL_DefaultResponse
- tsZCL_OctetString
- tsZCL_CharacterString
- tsZCL_ClusterCustomMessage
- tsZCL_ClusterInstance
- tsZCL_ClusterDefinition
- tsZCL_AttributeDefinition

13.2 Device Structures

This section presents the shared device structures for the SE devices supported by the SE API.

13.2.1 ESP/Metering Device (tsSE_EspMeterDevice)

The following `tsSE_EspMeterDevice` structure is the shared structure for a combined ESP/Metering device:

```
typedef struct
{
    tsZCL_EndPointDefinition sEndPoint;

    /* Cluster instances */
    tsSE_EspMeterDeviceClusterInstances sClusterInstance;

#ifdef CLD_BASIC
    /* Holds the attributes for the basic cluster */
    tsCLD_Basic sBasicCluster;

    /* Holds the status of the attributes in the basic cluster */
    /* sCLD_AS_Basic sBasicClusterAttributeStatus; */
#ifdef BASIC_CLIENT
    /* Holds the attributes for the basic cluster for whatever the IPD is 
monitoring */
    tsCLD_Basic sRemoteBasicCluster;
#endif
#endif

#ifdef CLD_SIMPLE_METERING

    /* Holds the attributes for the simple metering cluster */
    tsCLD_SimpleMetering sSimpleMeteringCluster;

    /* Holds the status of the attributes in the simple metering cluster */
    /*tsCLD_AS_SimpleMetering sSimpleMeteringClusterAttributeStatus;*/

    /*Event Address, Custom call back event, Custom call back message*/
    tsSM_CustomStruct sSimpleMeteringCustomDataStruct;

#endif

#ifdef CLD_TIME
#ifdef TIME_SERVER
    /* Pointer to the common time cluster */
    tsCLD_Time sTimeCluster;

    /* Pointer to the common time cluster attribute status */
    /*tsCLD_AS_Time sTimeClusterAttributeStatus;*/
#endif
#ifdef TIME_CLIENT
    /* Pointer to the common time cluster */
     tsCLD_Time sRemoteTimeCluster;
#endif
#endif

#ifdef CLD_PRICE
    /* price cluster */
    tsCLD_Price sPriceCluster;
    /* status of the attributes in the price cluster */
    /*tsCLD_AS_Price sPriceClusterAttributeStatus;*/

    /* custom data structures */
    tsSE_PriceCustomDataStructure sPriceCustomDataStructure;
    tsSE_PricePublishPriceRecord
asPublishPriceRecord[SE_PRICE_NUMBER_OF_SERVER_PRICE_RECORD_ENTRIES];
    uint8
au8RateLabel[SE_PRICE_NUMBER_OF_SERVER_PRICE_RECORD_ENTRIES][SE_PRICE_SER
VER_MAX_STRING_LENGTH];

#ifdef BLOCK_CHARGING
    /* Block Period */
    tsSE_PricePublishBlockPeriodRecord
asPublishBlockPeriodRecord[SE_PRICE_NUMBER_OF_SERVER_BLOCK_PERIOD_RECORD_
ENTRIES];
#endif /* BLOCK_CHARGING */

#ifdef PRICE_CONVERSION_FACTOR
    tsSE_PriceConversionFactorRecord
asPublishConversionFactorRecord[SE_PRICE_NUMBER_OF_CONVERSION_FACTOR_ENTR
IES];
#endif

#ifdef PRICE_CALORIFIC_VALUE
    tsSE_PriceCalorificValueRecord
asPublishCalorificValueRecord[SE_PRICE_NUMBER_OF_CALORIFIC_VALUE_ENTRIES]
;
#endif

#endif

#ifdef CLD_DRLC
    tsCLD_DRLC sDRLCCluster;
    /*tsCLD_AS_DRLC sDRLCClusterAttributeStatus;*/
    tsSE_DRLCCustomDataStructure sDRLCCustomDataStructure;
    tsSE_DRLCLoadControlEventRecord
asDRLCLoadControlEventRecord[SE_DRLC_NUMBER_OF_SERVER_LOAD_CONTROL_ENTRIE
S];
#endif
```

```
#ifdef CLD_KEY_ESTABLISHMENT
    tsCLD_KeyEstablishment sKeyEstablishmentCluster;
    /*tsCLD_AS_KeyEstablishment sKeyEstablishmentClusterAttributeStatus;*/
    tsSE_KECCustomDataStructure sKECCustomDataStructure;
#endif

#ifdef CLD_MC
    tsSE_MCCustomDataStructure sMessageCustomDataStructure;
    tsSE_MCDisplayMessageCommandPayloadRecord
asDisplayMessageCommandPayloadRecord[SE_MESSAGE_NUMBER_OF_SERVER_MESSAGE_
RECORD_ENTRIES];
    uint8
au8DisplayMessage[SE_MESSAGE_NUMBER_OF_SERVER_MESSAGE_RECORD_ENTRIES][SE_
MESSAGE_SERVER_MAX_STRING_LENGTH];
#endif

#if ((defined CLD_SIMPLE_METERING ) && (defined CLD_SM_SUPPORT_MIRROR))
    //Create an Array of Mirror Structure
    tsSE_Mirror    sSE_Mirrors[CLD_SM_NUMBER_OF_MIRRORS];
    //Create an Array for Mirrored Attribute set.
    tsSE_EspMirrorAttributeSet
sSE_MirrorAttributeSet[CLD_SM_NUMBER_OF_MIRRORS];
#endif

#ifdef CLD_OTA
   /* OTA cluster */
    tsCLD_AS_Ota sCLD_OTA;
    /* Status of OTA attributes in the cluster */
    /*tsCLD_AS_Ota_Status sOTAClusterAttributesStatus;*/

    /* custom data structures */
    /*tsOTA_Common sCLD_OTA_CustomDataStruct[SE_NUMBER_OF_ENDPOINTS];*/
    tsOTA_Common sCLD_OTA_CustomDataStruct;
#endif

#ifdef CLD_PREPAYMENT
    tsSE_PrepayDeviceInstance PrepayDeviceInfo;
#endif

#ifdef CLD_COMMISSIONING
    /* Pointer to the common time cluster */
    tsCLD_Commissioning sCommissioningCluster;
#endif

#if defined(CLD_IDENTIFY) && defined(IDENTIFY_SERVER)
    tsCLD_Identify                      sIdentifyCluster;
    tsCLD_IdentifyCustomDataStructure   sIdentifyCustomDataStructure;
#endif
} tsSE_EspMeterDevice;
```

There is a section of this structure for each cluster supported by the ESP/Metering Device. Each of these sections has one or more of the following elements:

- Pointer to the cluster
- Data structure(s) for the cluster

The section for each optional cluster is enabled by a corresponding enumeration defined in the **zcl_options.h** file (e.g. CLD_MC for the Messaging cluster). If a cluster is not defined in this header file then it does not feature in the above structure.

13.2.2 ESP (tsSE_EspDevice)

The following `tsSE_EspDevice` structure is the shared structure for an ESP:

```
typedef struct
{
    tsZCL_EndPointDefinition sEndPoint;

    /* Cluster instances */
    tsSE_EspDeviceClusterInstances sClusterInstance;

#ifdef CLD_BASIC
    /* Holds the attributes for the basic cluster */
    tsCLD_Basic sBasicCluster;

#ifdef BASIC_CLIENT
    /* Holds the attributes for the basic cluster for whatever the IPD is 
monitoring */
    tsCLD_Basic sRemoteBasicCluster;
#endif
#endif

#if ((defined CLD_SIMPLE_METERING)&&(defined SM_CLIENT))
    /* Holds the attributes for the simple metering cluster */
    tsCLD_SimpleMetering sSimpleMeteringCluster;
    /*Event Address, Custom call back event, Custom call back message*/
    tsSM_CustomStruct sSimpleMeteringCustomDataStruct;
#endif

#ifdef CLD_TIME
#ifdef TIME_SERVER
    /* Pointer to the common time cluster */
    tsCLD_Time sTimeCluster;
#endif

#ifdef TIME_CLIENT
    /* Pointer to the common time cluster */
     tsCLD_Time sRemoteTimeCluster;
#endif
```

```
#endif

#ifdef CLD_PRICE
    /* price cluster */
    tsCLD_Price sPriceCluster;
    /* custom data structures */
    tsSE_PriceCustomDataStructure sPriceCustomDataStructure;
    tsSE_PricePublishPriceRecord 
asPublishPriceRecord[SE_PRICE_NUMBER_OF_SERVER_PRICE_RECORD_ENTRIES];
    uint8 
au8RateLabel[SE_PRICE_NUMBER_OF_SERVER_PRICE_RECORD_ENTRIES][SE_PRICE_SERVER
_MAX_STRING_LENGTH];

#ifdef BLOCK_CHARGING
    /* Block Period */
    tsSE_PricePublishBlockPeriodRecord 
asPublishBlockPeriodRecord[SE_PRICE_NUMBER_OF_SERVER_BLOCK_PERIOD_RECORD_ENT
RIES];
#endif /* BLOCK_CHARGING */
#ifdef PRICE_CONVERSION_FACTOR
    tsSE_PriceConversionFactorRecord 
asPublishConversionFactorRecord[SE_PRICE_NUMBER_OF_CONVERSION_FACTOR_ENTRIES
];
#endif

#ifdef PRICE_CALORIFIC_VALUE
    tsSE_PriceCalorificValueRecord 
asPublishCalorificValueRecord[SE_PRICE_NUMBER_OF_CALORIFIC_VALUE_ENTRIES];
#endif
#endif

#ifdef CLD_DRLC
    tsCLD_DRLC sDRLCCluster;
    tsSE_DRLCCustomDataStructure sDRLCCustomDataStructure;
    tsSE_DRLCLoadControlEventRecord 
asDRLCLoadControlEventRecord[SE_DRLC_NUMBER_OF_SERVER_LOAD_CONTROL_ENTRIES];
#endif

#ifdef CLD_KEY_ESTABLISHMENT
    tsCLD_KeyEstablishment sKeyEstablishmentCluster;
    tsSE_KECCustomDataStructure sKECCustomDataStructure;
#endif

#ifdef CLD_MC
    tsSE_MCCustomDataStructure sMessageCustomDataStructure;
    tsSE_MCDisplayMessageCommandPayloadRecord 
asDisplayMessageCommandPayloadRecord[SE_MESSAGE_NUMBER_OF_SERVER_MESSAGE_REC
ORD_ENTRIES];
    uint8 
au8DisplayMessage[SE_MESSAGE_NUMBER_OF_SERVER_MESSAGE_RECORD_ENTRIES][SE_MES
SAGE_SERVER_MAX_STRING_LENGTH];
#endif
```

```
#if ((defined CLD_SIMPLE_METERING ) && (defined CLD_SM_SUPPORT_MIRROR))
    //Create an Array of Mirror Structure
    tsSE_Mirror    sSE_Mirrors[CLD_SM_NUMBER_OF_MIRRORS];
    //Create an Array for Mirrored Attribute set.
    tsSE_EspMirrorAttributeSet 
sSE_MirrorAttributeSet[CLD_SM_NUMBER_OF_MIRRORS];
#endif

#ifdef CLD_OTA
   /* OTA cluster */
    tsCLD_AS_Ota sCLD_OTA;
    /* custom data structures */
    tsOTA_Common sCLD_OTA_CustomDataStruct;
#endif

#ifdef CLD_PREPAYMENT
    tsSE_PrepayDeviceInstance PrepayDeviceInfo;
#endif

#ifdef CLD_COMMISSIONING
    /* Pointer to the common time cluster */
    tsCLD_Commissioning sCommissioningCluster;
#endif

#if defined(CLD_IDENTIFY) && defined(IDENTIFY_SERVER)
    tsCLD_Identify                      sIdentifyCluster;
    tsCLD_IdentifyCustomDataStructure   sIdentifyCustomDataStructure;
#endif
} tsSE_EspDevice;
```

There is a section of this structure for each cluster supported by the ESP. Each of these sections has one or more of the following elements:

- Pointer to the cluster
- Data structure(s) for the cluster

The section for each optional cluster is enabled by a corresponding enumeration defined in the **zcl_options.h** file (e.g. CLD_MC for the Messaging cluster). If a cluster is not defined in this header file then it does not feature in the above structure.

13.2.3 Metering Device (tsSE_MeterDevice)

The following `tsSE_MeterDevice` structure is the shared structure for a Metering Device:

```
typedef struct
{
    tsZCL_EndPointDefinition sEndPoint;

    /* Cluster instances */
    tsSE_MeterDeviceClusterInstances sClusterInstance;

#ifdef CLD_BASIC
    /* Holds the attributes for the basic cluster */
    tsCLD_Basic sBasicCluster;
#ifdef BASIC_CLIENT
    /* Holds the attributes for the basic cluster for whatever the IPD is
monitoring */
    tsCLD_Basic sRemoteBasicCluster;
#endif
#endif
#ifdef CLD_SIMPLE_METERING
    /* Holds the attributes for the simple metering cluster */
    tsCLD_SimpleMetering sSimpleMeteringCluster;
    /*Event Address, Custom call back event, Custom call back message*/
    tsSM_CustomStruct sSimpleMeteringCustomDataStruct;
#endif

#if (defined CLD_TIME && defined TIME_CLIENT)
    /* Pointer to the common time cluster */
    tsCLD_Time sTimeCluster;
#endif

#ifdef CLD_PRICE
    /* price cluster */
    tsCLD_Price sPriceCluster;

    /* custom data structures */
    tsSE_PriceCustomDataStructure sPriceCustomDataStructure;
    tsSE_PricePublishPriceRecord
asPublishPriceRecord[SE_PRICE_NUMBER_OF_CLIENT_PRICE_RECORD_ENTRIES];
```

```
    uint8
au8RateLabel[SE_PRICE_NUMBER_OF_CLIENT_PRICE_RECORD_ENTRIES][SE_PRICE_SER
VER_MAX_STRING_LENGTH];

#ifdef BLOCK_CHARGING
    /* Block Period */
    tsSE_PricePublishBlockPeriodRecord
asPublishBlockPeriodRecord[SE_PRICE_NUMBER_OF_CLIENT_BLOCK_PERIOD_RECORD_
ENTRIES];
#endif /* BLOCK_CHARGING */
#ifdef PRICE_CONVERSION_FACTOR
    tsSE_PriceConversionFactorRecord
asPublishConversionFactorRecord[SE_PRICE_NUMBER_OF_CONVERSION_FACTOR_ENTR
IES];
#endif

#ifdef PRICE_CALORIFIC_VALUE
    tsSE_PriceCalorificValueRecord
asPublishCalorificValueRecord[SE_PRICE_NUMBER_OF_CALORIFIC_VALUE_ENTRIES]
;
#endif
#endif

#ifdef CLD_KEY_ESTABLISHMENT
    /* key establishment cluster */
    tsCLD_KeyEstablishment sKeyEstablishmentCluster;
    tsSE_KECCustomDataStructure sKECCustomDataStructure;
#endif

#ifdef CLD_MC
    tsSE_MCCustomDataStructure sMessageCustomDataStructure;
    tsSE_MCDisplayMessageCommandPayloadRecord
asDisplayMessageCommandPayloadRecord[SE_MESSAGE_NUMBER_OF_CLIENT_MESSAGE_
RECORD_ENTRIES];
    uint8
au8DisplayMessage[SE_MESSAGE_NUMBER_OF_CLIENT_MESSAGE_RECORD_ENTRIES][SE_
MESSAGE_SERVER_MAX_STRING_LENGTH];
#endif

#ifdef CLD_OTA
   /* OTA cluster */
    tsCLD_AS_Ota sCLD_OTA;
    /* custom data structures */
    tsOTA_Common sCLD_OTA_CustomDataStruct;
#endif

#ifdef CLD_PREPAYMENT
    tsSE_PrepayDeviceInstance PrepayDeviceInfo;
#endif

#ifdef CLD_COMMISSIONING
    tsCLD_Commissioning sCommissioningCluster;
```

```
#endif

#if defined(CLD_IDENTIFY) && defined(IDENTIFY_SERVER)
    tsCLD_Identify                      sIdentifyCluster;
    tsCLD_IdentifyCustomDataStructure   sIdentifyCustomDataStructure;
#endif
} tsSE_MeterDevice;
```

There is a section of this structure for each cluster supported by the Metering Device. Each of these sections has one or more of the following elements:

- Pointer to the cluster
- Data structure(s) for the cluster

The section for each optional cluster is enabled by a corresponding enumeration defined in the **zcl_options.h** file (e.g. CLD_MC for the Messaging cluster). If a cluster is not defined in this header file then it does not feature in the above structure.

13.2.4 IPD (tsSE_IPDDevice)

The following `tsSE_IPDDevice` structure is the shared structure for an IPD:

```
typedef struct
{
    tsZCL_EndPointDefinition sEndPoint;

    /* Cluster instances */
    tsSE_IPDDeviceClusterInstances sClusterInstance;

    /* Holds the attributes for the basic cluster for the IPD */
    tsCLD_Basic sLocalBasicCluster;

    /* Holds the status of the attributes in the basic cluster */
    /*tsCLD_AS_Basic sLocalBasicClusterAttributeStatus;*/

    /* Holds the attributes for the basic cluster for whatever the IPD is 
monitoring */
    tsCLD_Basic sRemoteBasicCluster;

    /* Holds the status of the attributes in the basic cluster */
    /*tsCLD_AS_Basic sRemoteBasicClusterAttributeStatus;*/
#ifdef CLD_SIMPLE_METERING
    /* Holds the attributes for the simple metering cluster */
    tsCLD_SimpleMetering sSimpleMeteringCluster;

    /* Holds the status of the attributes in the simple metering cluster */
    /*tsCLD_AS_SimpleMetering sSimpleMeteringClusterAttributeStatus;*/

    /*Event Address, Custom call back event, Custom call back message*/
    tsSM_CustomStruct sSimpleMeteringCustomDataStruct;
#endif

#if (defined CLD_TIME && defined TIME_CLIENT)
    /* Pointer to the common time cluster */
    tsCLD_Time sTimeCluster;
#endif

#ifdef CLD_PRICE
    /* the price cluster */
    tsCLD_Price sPriceCluster;

    /* the common time cluster attribute status */
    /*tsCLD_AS_Price sPriceClusterAttributeStatus;*/

     /* custom data structures */
    tsSE_PriceCustomDataStructure sPriceCustomDataStructure;
    tsSE_PricePublishPriceRecord 
asPublishPriceRecord[SE_PRICE_NUMBER_OF_CLIENT_PRICE_RECORD_ENTRIES];
```

```
    uint8
au8RateLabel[SE_PRICE_NUMBER_OF_CLIENT_PRICE_RECORD_ENTRIES][SE_PRICE_SERVER
_MAX_STRING_LENGTH];

#ifdef BLOCK_CHARGING
    /* Block Period */
    tsSE_PricePublishBlockPeriodRecord
asPublishBlockPeriodRecord[SE_PRICE_NUMBER_OF_CLIENT_BLOCK_PERIOD_RECORD_ENT
RIES];
#endif /* BLOCK_CHARGING */
#ifdef PRICE_CONVERSION_FACTOR
    tsSE_PriceConversionFactorRecord
asPublishConversionFactorRecord[SE_PRICE_NUMBER_OF_CONVERSION_FACTOR_ENTRIES
];
#endif

#ifdef PRICE_CALORIFIC_VALUE
    tsSE_PriceCalorificValueRecord
asPublishCalorificValueRecord[SE_PRICE_NUMBER_OF_CALORIFIC_VALUE_ENTRIES];
#endif
#endif

#ifdef CLD_DRLC
    tsCLD_DRLC sDRLCCluster;
    /*tsCLD_AS_DRLC sDRLCClusterAttributeStatus;*/
    /* custom data structures */
    tsSE_DRLCCustomDataStructure sDRLCCustomDataStructure;
    tsSE_DRLCLoadControlEventRecord
asDRLCLoadControlEventRecord[SE_DRLC_NUMBER_OF_CLIENT_LOAD_CONTROL_ENTRIES];
#endif

#ifdef CLD_KEY_ESTABLISHMENT
    /* key establishment cluster */
    tsCLD_KeyEstablishment sKeyEstablishmentCluster;
    /* status of the attributes in the price cluster */
    /*tsCLD_AS_KeyEstablishment sKeyEstablishmentClusterAttributeStatus;*/
    tsSE_KECCustomDataStructure sKECCustomDataStructure;
#endif

#ifdef CLD_MC
    tsSE_MCCustomDataStructure sMessageCustomDataStructure;
    tsSE_MCDisplayMessageCommandPayloadRecord
asDisplayMessageCommandPayloadRecord[SE_MESSAGE_NUMBER_OF_CLIENT_MESSAGE_REC
ORD_ENTRIES];
    uint8
au8DisplayMessage[SE_MESSAGE_NUMBER_OF_CLIENT_MESSAGE_RECORD_ENTRIES][SE_MES
SAGE_SERVER_MAX_STRING_LENGTH];
#endif

#ifdef CLD_OTA

   /* OTA cluster */
```

```
    tsCLD_AS_Ota sCLD_OTA;
    /* Status of OTA attributes in the cluster */
    /*tsCLD_AS_Ota_Status 
sOTAClusterAttributesStatus[SE_NUMBER_OF_ENDPOINTS];*/
    /* custom data structures */
    /*tsOTA_Common sCLD_OTA_CustomDataStruct[SE_NUMBER_OF_ENDPOINTS];*/
    tsOTA_Common sCLD_OTA_CustomDataStruct;
#endif

#ifdef CLD_PREPAYMENT
    tsSE_PrepayDeviceInstance PrepayDeviceInfo;
#endif

#ifdef CLD_COMMISSIONING
    tsCLD_Commissioning sCommissioningCluster;
#endif

#if defined(CLD_IDENTIFY) && defined(IDENTIFY_SERVER)
    tsCLD_Identify                      sIdentifyCluster;
    tsCLD_IdentifyCustomDataStructure   sIdentifyCustomDataStructure;
#endif

#if defined(CLD_IDENTIFY) && defined(IDENTIFY_CLIENT)
    tsCLD_Identify                      sIdentifyClientCluster;
    tsCLD_IdentifyCustomDataStructure   sIdentifyClientCustomDataStructure;
#endif
} tsSE_IPDDevice;
```

There is a section of this structure for each cluster supported by the IPD. Each of these sections has one or more of the following elements:

- Pointer to the cluster
- Data structure(s) for the cluster

The section for each optional cluster is enabled by a corresponding enumeration defined in the **zcl_options.h** file (e.g. CLD_MC for the Messaging cluster). If a cluster is not defined in this header file then it does not feature in the above structure.

13.2.5 Range Extender (tsSE_RangeExtDevice)

The following `tsSE_RangeExtDevice` structure is the shared structure for a Range Extender:

```
typedef struct
{
    tsZCL_EndPointDefinition sEndPoint;

    /* Cluster instances */
    tsSE_RangeExtDeviceClusterInstances sClusterInstance;

#ifdef CLD_BASIC
    /* Holds the attributes for the basic cluster */
    tsCLD_Basic sBasicCluster;

    /* Holds the status of the attributes in the basic cluster */
    //tsCLD_AS_Basic sLocalBasicClusterAttributeStatus;
#ifdef BASIC_CLIENT
    /* Holds the attributes for the basic cluster for whatever the IPD is 
monitoring */
    tsCLD_Basic sRemoteBasicCluster;

    /* Holds the status of the attributes in the basic cluster */
    //tsCLD_AS_Basic sRemoteBasicClusterAttributeStatus;
    #endif
#endif

#ifdef CLD_KEY_ESTABLISHMENT

    /* key establishment cluster */
    tsCLD_KeyEstablishment sKeyEstablishmentCluster;
    /* status of the attributes in the price cluster */
    //tsCLD_AS_KeyEstablishment sKeyEstablishmentClusterAttributeStatus;
    tsSE_KECCustomDataStructure sKECCustomDataStructure;
#endif

#ifdef CLD_OTA

   /* OTA cluster */
    tsCLD_AS_Ota sCLD_OTA;
    /* custom data structures */
    tsOTA_Common sCLD_OTA_CustomDataStruct;
#endif
} tsSE_RangeExtDevice;
```

13.3 Event Structure and Enumerations

The following event-related structures are defined in the ZigBee Cluster Library and are detailed in the *ZCL User Guide (JN-UG-3077)*:

- tsZCL_CallBackEvent (event structure)
- teZCL_CallBackEventType (event type enumerations)

13.4 ZCL Enumerations

The following sets of enumerations are defined in the ZigBee Cluster Library and are detailed in the *ZCL User Guide (JN-UG-3077)*:

- teZCL_Status (function return codes)
- eZCL_AddressMode (addressing modes)
- ZPS_teAplAfBroadcastMode (broadcast modes)
- teZCL_ZCLAttributeType (attribute types)
- teZCL_RequestStatus (request status)
- teZCL_CommandStatus (command status)
- teZCL_ZCLSendSecurity (security level)

13.5 ZigBee Network Parameters

The ZigBee network parameters are configurable in the ZPS Configuration Editor and are described in the *ZigBee PRO Stack User Guide (JN-UG-3048)*. Some of these parameters require specific settings for Smart Energy applications. These parameters are listed in the table below along with the required settings (the paths and parameter names are as used in the ZPS Configuration Editor).

Path *	**Parameter**	**Required Value**
Co-ordinator > Properties	*Initial Security Key*	Random (Co-ordinator only)
Co-ordinator > ZDO Configuration > End Device Bind Server > Properties	*Timeout*	60 (Co-ordinator only)
Any Device > Advanced Properties	*APS Inter-frame Delay*	50
	APS Max Window Size	1
	APS Use Insecure Join	FALSE
	APS Security Timeout Period	1000 (default)

Table 52: Network Parameter Settings for Smart Energy

* Path is obtained in the ZPS Configuration Editor by selecting a ZigBee device (and possibly one or more sub-entries) in the tree and then clicking the **Properties** or **Advanced Properties** button - the parameters then appear in the bottom panel of the editor.

Part IV: Appendices

A. Supported Clusters and Attributes

The General and Smart Energy clusters that are included in the ZigBee Smart Energy Profile Specification 1.1.1 are listed in Section 2.3. Not all of these clusters are certifiable in SE 1.1.1 (07-5356-17) or earlier and supported by the NXP ZigBee PRO Smart Energy API. The supported clusters are listed in the table below.

Category	Cluster	Cluster ID
General (ZCL)	Basic	0x0000
	Identify	0x0003
	Time	0x000A
	Commissioning	0x0015
	Over-the-Air (OTA) Upgrade	0x0019
Smart Energy	Price	0x0700
	Demand-Response and Load Control	0x0701
	(Simple) Metering	0x0702
	Messaging	0x0703
	Key Establishment	0x0800

Table 53: SE Profile Clusters

The attributes supported by each Smart Energy cluster are indicated in the sections below.

A.1 Price Cluster Attributes

Price Cluster Server Attributes

- Tier Label attribute set
  - `Tier<x>PriceLabel` (where <x>=1, 2,... 15)
- Block Threshold attribute set
  - `Block<x>Threshold` (where <x>=1, 2,... 15)
- Block Period attribute set
  - `StartofBlockPeriod`
  - `BlockPeriodDuration` (minutes)
  - `ThresholdMultiplier`
  - `ThresholdDivisor`
- Commodity attribute set
  - `CommodityType`
  - `StandingCharge`

    - `ConversionFactor`
    - `ConversionFactorTrailingDigit`
    - `CalorificValue`
    - `CalorificValueTrailingDigit`
    - `CalorificValueUnit`
- Block Price Information attribute set
    - `NoTierBlock<x>Price` (where <x>=1, 2,... 16)
    - `Tier1Block<x>Price` (where <x>=1, 2,... 16)
    - `Tier2Block<x>Price` (where <x>=1, 2,... 16)
    - `Tier3Block<x>Price` (where <x>=1, 2,... 16)
    - `Tier4Block<x>Price` (where <x>=1, 2,... 16)
    - `Tier5Block<x>Price` (where <x>=1, 2,... 16)
    - `Tier6Block<x>Price` (where <x>=1, 2,... 16)
    - `Tier7Block<x>Price` (where <x>=1, 2,... 16)
    - `Tier8Block<x>Price` (where <x>=1, 2,... 16)
    - `Tier9Block<x>Price` (where <x>=1, 2,... 16)
    - `Tier10Block<x>Price` (where <x>=1, 2,... 16)
    - `Tier11Block<x>Price` (where <x>=1, 2,... 16)
    - `Tier12Block<x>Price` (where <x>=1, 2,... 16)
    - `Tier13Block<x>Price` (where <x>=1, 2,... 16)
    - `Tier14Block<x>Price` (where <x>=1, 2,... 16)
    - `Tier15Block<x>Price` (where <x>=1, 2,... 16)
- Billing Period Information attribute set
    - `StartOfBillingPeriod`
    - `BillingPeriodDuration`

Price Cluster Client Attributes

- `u8ClientIncreaseRandomize`
- `u8ClientDecreaseRandomize`
- `e8ClientCommodityType`

A.2 Demand-Response and Load Control Cluster Attributes

Note: There are no server attributes in the Demand-Response and Load Control cluster.

DRLC Cluster Client Attributes

- `UtilityEnrolmentGroup`
- `StartRandomizeMinutes`
- `StopRandomizeMinutes`
- `DeviceClassValue`

A.3 (Simple) Metering Cluster Attributes

Note: There are no client attributes in the Simple Metering cluster.

Metering Cluster Server Attributes

- Reading Information attribute set
  - `CurrentSummationDelivered`
  - `CurrentSummationReceived`
  - `CurrentMaxDemandDelivered`
  - `CurrentMaxDemandReceived`
  - `DFTSummation`
  - `Daily FreezeTime`
  - `PowerFactor`
  - `ReadingSnapShotTime`
  - `CurrentMaxDemandDeliveredTime`
  - `CurrentMaxDemandReceivedTime`
  - `DefaultUpdatePeriod`
  - `FastPollUpdatePeriod`
  - `CurrentBlockPeriodConsumptionDelivered`
  - `DailyConsumptionTarget`
  - `CurrentBlock`
  - `ProfileIntervalPeriod`

- `IntervalReadReportingPeriod`
- `PresetReadingTime`
- `VolumePerReport`
- `FlowRestriction`
- `Supply Status`
- `CurrentInletEnergyCarrierSummation`
- `CurrentOutletEnergyCarrierSummation`
- `InletTemperature`
- `OutletTemperature`
- `ControlTemperature`
- `CurrentInletEnergyCarrierDemand`
- `CurrentOutletEnergyCarrierDemand`

- TOU Information attribute set
  - `CurrentTier<x>SummationDelivered` (where <x>=1, 2,... 16)
  - `CurrentTier<x>SummationReceived` (where <x>=1, 2,... 16)
- Meter Status attribute set
  - `Status`
  - `RemainingBatteryLife`
  - `HoursInOperation`
  - `HoursInFault`
- Formatting Attribute Set
  - `UnitofMeasure`
  - `Multiplier`
  - `Divisor`
  - `SummationFormatting`
  - `DemandFormatting`
  - `HistoricalConsumptionFormatting`
  - `MeteringDeviceType`
  - `SiteID`
  - `MeterSerialNumber`
  - `EnergyCarrierUnitOfMeasure`
  - `EnergyCarrierSummationFormatting`
  - `EnergyCarrierDemandFormatting`
  - `TemperatureUnitOfMeasure`
  - `TemperatureFormatting`

- Historical Consumption attribute set
  - `InstantaneousDemand`
  - `CurrentDayConsumptionDelivered`
  - `CurrentDayConsumptionReceived`
  - `PreviousDayConsumptionDelivered`
  - `PreviousDayConsumptionReceived`
  - `CurrentPartialProfileIntervalStartTimeDelivered`
  - `CurrentPartialProfileIntervalStartTimeReceived`
  - `CurrentPartialProfileIntervalValueDelivered`
  - `CurrentPartialProfileIntervalValueReceived`
  - `CurrentDayMaxPressure`
  - `CurrentDayMinPressure`
  - `PreviousDayMaxPressure`
  - `PreviousDayMinPressure`
  - `CurrentDayMaxDemand`
  - `PreviousDayMaxDemand`
  - `CurrentMonthMaxDemand`
  - `CurrentYearMaxDemand`
  - `CurrentDayMaxEnergyCarrierDemand`
  - `PreviousDayMaxEnergyCarrierDemand`
  - `CurrentMonthMaxEnergyCarrierDemand`
  - `CurrentMonthMinEnergyCarrierDemand`
  - `CurrentYearMaxEnergyCarrierDemand`
  - `CurrentYearMinEnergyCarrierDemand`
- Load Profile Configuration attribute set
  - `MaxNumberOfPeriodsDelivered`
- Supply Limit attribute set
  - `CurrentDemandDelivered`
  - `DemandLimit`
  - `DemandIntegration Period`
  - `NumberOfDemandSubintervals`
- Block Information attribute set
  - `CurrentNoTierBlock<x>SummationDelivered`
    (where <x>=1, 2,... 16)
  - `CurrentTier1Block<x>SummationDelivered`
    (where <x>=1, 2,... 16)

  - `CurrentTier2Block<x>SummationDelivered` (where <x>=1, 2,... 16)
  - `CurrentTier3Block<x>SummationDelivered` (where <x>=1, 2,... 16)
  - `CurrentTier4Block<x>SummationDelivered` (where <x>=1, 2,... 16)
  - `CurrentTier5Block<x>SummationDelivered` (where <x>=1, 2,... 16)
  - `CurrentTier6Block<x>SummationDelivered` (where <x>=1, 2,... 16)
  - `CurrentTier7Block<x>SummationDelivered` (where <x>=1, 2,... 16)
  - `CurrentTier8Block<x>SummationDelivered` (where <x>=1, 2,... 16)
  - `CurrentTier9Block<x>SummationDelivered` (where <x>=1, 2,... 16)
  - `CurrentTier10Block<x>SummationDelivered` (where <x>=1, 2,... 16)
  - `CurrentTier11Block<x>SummationDelivered` (where <x>=1, 2,... 16)
  - `CurrentTier12Block<x>SummationDelivered` (where <x>=1, 2,... 16)
  - `CurrentTier13Block<x>SummationDelivered` (where <x>=1, 2,... 16)
  - `CurrentTier14Block<x>SummationDelivered` (where <x>=1, 2,... 16)
  - `CurrentTier15Block<x>SummationDelivered` (where <x>=1, 2,... 16)

- Alarms attribute set
  - `GenericAlarmMask`
  - `ElectricityAlarmMask`
  - `Generic Flow/PressureAlarmMask`
  - `Water SpecificAlarmMask`
  - `Heat andCoolingSpecificAlarmMask`
  - `Gas SpecificAlarmMask`

A.4 Messaging Cluster Attributes

The Messaging cluster does not have any attributes.

A.5 Key Establishment Cluster Attributes

Key Establishment Cluster Server Attributes

- `KeyEstablishmentSuite`

Key Establishment Cluster Client Attributes

- `KeyEstablishmentSuite`

B. Custom Endpoints

An SE device and its associated clusters can be registered on an endpoint using the relevant device registration function, from those listed and described in Chapter 12. However, it is also possible to set up a custom endpoint which supports selected clusters (rather than a whole SE device and all of its associated clusters). Custom endpoints are particularly useful when using multiple endpoints on a single node - for example, the first endpoint may support a complete SE device (such as an IPD) while one or more custom endpoints are used to support selected clusters.

B.1 SE Devices and Endpoints

When using custom endpoints, it is important to note the difference between the following SE ‘devices’:

- **Physical device:** This is the physical entity which is the SE network node
- **Logical SE device:** This is a software entity which implements a specific set of SE functionality on the node, e.g. Metering device

An SE network node may contain multiple endpoints, where one endpoint is used to represent the ‘physical device’ and other endpoints are used to support ‘logical SE devices’. The following rules apply to cluster instances on endpoints:

- All cluster instances relating to a single ‘logical SE device’ must reside on a single endpoint.
- The Basic cluster and Key Establishment cluster relate to the ‘physical device’ rather than a ‘logical SE device’ instance. For each of these clusters, there can be only one cluster server for the entire node, which can be implemented in either of the following ways:
    - A single cluster instance on a dedicated ‘physical device’ endpoint (instances for both clusters can reside on this endpoint)
    - A separate cluster instance on each ‘logical SE device’ endpoint, but each cluster instance must use the same `tsZCL_ClusterInstance` structure (and the same attribute values)

B.2 Cluster Creation Functions

For each cluster, a creation function is provided which creates an instance of the cluster on an endpoint. These functions are as follows:

- Basic: **eCLD_BasicCreateBasic()**
- Identify: **eCLD_IdentifyCreateIdentify()**
- Time: **eCLD_TimeCreateTime()**
- Price: **eSE_PriceCreate()**
- Messaging: **eSE_MCCreate()**
- Simple Metering: **eSE_SMCreate()**
- Demand-Response and Load Control: **eSE_DRLCCreate()**
- Key Establishment: **eSE_KECCreate()**

More than one of the above functions can be called for the same endpoint in order to create multiple cluster instances on the endpoint.

Note: No more than one server instance and one client instance of a given cluster can be created on a single endpoint (e.g. one Identify cluster server and one Identify cluster client, but no further Identify cluster instances).

The creation functions for clusters from the ZCL are described in the *ZCL User Guide (JN-UG-3077)*. The creation functions for the remaining SE-specific clusters are described in the chapters for the corresponding clusters in this manual.

B.3 Custom Endpoint Set-up

In order to set up a custom endpoint (supporting selected clusters), you must do the following in your application code:

1. Create a structure for the custom endpoint containing details of the cluster instances and attributes supported - see Appendix B.3.1 and Appendix B.3.2.
2. Initialise the fields of the `tsZCL_EndPointDefinition` structure for the endpoint.
3. Call the relevant cluster creation function(s) for the cluster(s) to be supported on the endpoint - see Appendix B.2.
4. Call the ZCL function **eZCL_Register()** for the endpoint.

B.3.1 Custom Endpoint Structure

In your application code, to set up a custom endpoint you must create a structure containing details of the cluster instances and attributes to be supported on the endpoint. This structure must include the following:

- A definition of the custom endpoint through a `tsZCL_EndPointDefinition` structure - for example:

  ```
  tsZCL_EndPointDefinition sEndPoint
  ```

- A structure containing a set of `tsZCL_ClusterInstance` structures for the supported cluster instances - for example:

  ```
  typedef struct
  {
      tsZCL_ClusterInstance sBasicServer;
      tsZCL_ClusterInstance sBasicClient;
      tsZCL_ClusterInstance sSimpleMeteringClient;
      tsZCL_ClusterInstance sDRLCClient;
      tsZCL_ClusterInstance sPriceClient;
      tsZCL_ClusterInstance sKeyEstablishmentClient;
      tsZCL_ClusterInstance sTimeClient;
      tsZCL_ClusterInstance sIdentifyClient;
  } tsSE_AppCustomDeviceClusterInstances
  ```

  For each cluster instance that is not shared with another endpoint, the following should be specified via the relevant `tsZCL_ClusterInstance` structure:

  - Attribute definitions, if any - for example, the `tsCLD_Basic` structure for the Basic cluster
  - Custom data structures, if any - for example, the `tsSM_CustomStruct` structure for the Simple Metering cluster
  - Memory for tables or any other resources, if required by the cluster creation function

An example of a custom endpoint structure is provided in the example code fragment in Appendix B.3.2.

Note: If a custom endpoint is to co-exist with a device endpoint (as in the example in Appendix B.3.2), the endpoints can share the structures for the clusters that they have in common. Therefore, it is not necessary to define these cluster structures for the custom endpoint, since they already exist for the device endpoint.

B.3.2 Example Code for Custom Endpoint

The code fragment below illustrates how to set up a custom endpoint for an IPD in addition to an endpoint for a standard IPD. The two IPD endpoints use the same clusters and can share the cluster structures but, in this case, the custom IPD uses its own Price and DRLC structures.

```
/* The following variable will be used for registering endpoint1 */
tsSE_IPDDevices SE_IPDDevice;

/* Cluster instances for endpoint2 */
typedef struct
{
    tsZCL_ClusterInstance sBasicServer;
    tsZCL_ClusterInstance sBasicClient;
    tsZCL_ClusterInstance sSimpleMeteringClient;
    tsZCL_ClusterInstance sDRLCClient;
    tsZCL_ClusterInstance sPriceClient;
    tsZCL_ClusterInstance sDRLCClient;
    tsZCL_ClusterInstance sKeyEstablishmentClient;
    tsZCL_ClusterInstance sTimeClient;
    tsZCL_ClusterInstance sIdentifyClient;
} tsAPP_IPDDeviceClusterInstances;

typedef struct
{
    /* Endpoint details*/
    tsZCL_EndPointDefinition sEndPoint;

    /* Cluster instances */
    tsAPP_IPDDeviceClusterInstances sClusterInstance;

/* When setting up a second endpoint, it is not necessary to define all cluster-related structures as these
can be shared between endpoint1 and endpoint2. This is up to the application. In this example, only
structures for the Price and DRLC clusters are specifically set up for endpoint2. */

    /* Price cluster */
#ifdef PRICE_CLIENT
    tsCLD_Price sPriceCluster;
    tsSE_PriceCustomDataStructure sPriceCustomDataStructure;
    tsSE_PricePublishPriceRecord asPublishPriceRecord [SE_PRICE_NUMBER_OF_CLIENT_PRICE_RECORD_ENTRIES];
    uint8 au8RateLabel
[SE_PRICE_NUMBER_OF_CLIENT_PRICE_RECORD_ENTRIES][SE_PRICE_SERVER_MAX_STRING_LENGTH];
#endif

#ifdef DRLC_CLIENT

   /* DRLC */
    tsCLD_DRLC sDRLCCluster;
    tsSE_DRLCCustomDataStructure sDRLCCustomDataStructure;
    tsSE_DRLCLoadControlEventRecord asDRLCLoadControlEventRecord
[SE_DRLC_NUMBER_OF_CLIENT_LOAD_CONTROL_ENTRIES];
#endif
}tsAPP_IpdDevice;

/* Define endpoint instance in Application global area as follows */
tsAPP_IpdDevice sIPDDevice;

/* Registration function */
teZCL_Status eRegisterCustomIPDNode(void )
{
    uint8 i;
    teZCL_Status eZCL_Status;

    /* First register the endpoint using the SE-defined endpoint function */
    eZCL_Status = eSE_RegisterIPDEndPoint(IPD_BASE_LOCAL_EP, &cbZCL_EndpointCallback, &sSE_IPDDevice);
    if(E_ZCL_SUCCESS != eZCL_Status)
    {
        return eZCL_Status;
    }
    sSE_IPDDevice.sDRLCCluster.u16DeviceClassValue = E_SE_DRLC_SMART_APPLIANCES_BIT;

    /* Other endpoints share the cluster data with endpoint1 for the Basic,
     * Time, KEC, SM, and Identify clusters, but not for the DRLC and
     * Price clusters */

        /* Fill in end point details */
        sIPDDevice.sEndPoint.u8EndPointNumber = IPD_BASE_LOCAL_EP + 1;
        sIPDDevice.sEndPoint.u16ManufacturerCode = SE_MANUFACTURER_CODE;
        sIPDDevice.sEndPoint.u16ProfileEnum = SE_PROFILE_ID;
        sIPDDevice.sEndPoint.bIsManufacturerSpecificProfile = FALSE;
        sIPDDevice.sEndPoint.u16NumberOfClusters =
           sizeof(tsAPP_IPDDeviceClusterInstances) / sizeof(tsZCL_ClusterInstance);;
        sIPDDevice.sEndPoint.psClusterInstance =
           (tsZCL_ClusterInstance*)&sIPDDevice.sClusterInstance;
        sIPDDevice.sEndPoint.bDisableDefaultResponse = TRUE;
        sIPDDevice.sEndPoint.pCallBackFunctions = cbZCL_EndpointCallback;

        /* Create Basic server. Share the cluster data from endpoint1 */
#ifdef BASIC_SERVER
        if (eCLD_BasicCreateBasic(
                       &sIPDDevice.sClusterInstance.sBasicServer,
                       TRUE,
                       &sCLD_Basic,
                       &sSE_IPDDevice.sLocalBasicCluster,
                       &au8AppBasicServerAttributeControlBits[0]
                       ) != E_ZCL_SUCCESS)
            {
                return E_ZCL_FAIL;
            }
#else
sIPDDevice .sClusterInstance. sBasicServer.psClusterDefinition = &sCLD_BasicDummy;
#endif

        /* Create Basic client. Share the cluster data from endpoint1 */
#ifdef BASIC_CLIENT

        if (eCLD_BasicCreateBasic(
                       &sIPDDevice.sClusterInstance.sBasicClient,
                       FALSE,
                       &sCLD_Basic,
                       &sSE_IPDDevice.sRemoteBasicCluster,
                       &au8AppBasicClientAttributeControlBits[0]
                       ) != E_ZCL_SUCCESS)
           {
               return E_ZCL_FAIL;
           }

#else
      sIPDDevice .sClusterInstance. sBasicClient.psClusterDefinition = &sCLD_BasicDummy;
#endif

#ifdef SM_CLIENT
        /* Create SM client. Share the cluster data from endpoint1 */
        if (eSE_SMCreate(
           IPD_BASE_LOCAL_EP + 1,
           FALSE,
           &au8AppSMClientAttributeControlBits[0],
           &sIPDDevice.sClusterInstance.sSimpleMeteringClient,
           &sCLD_SimpleMetering,
           &sSE_IPDDevice.sSimpleMeteringCustomDataStruct, /* Information shared with endpoint1 */
           &sSE_IPDDevice.sSimpleMeteringCluster)  /* Information shared with endpoint1 */
            != E_ZCL_SUCCESS)
        {
                      return E_ZCL_FAIL;
        }

#else
      sIPDDevice .sClusterInstance. sSimpleMeteringClient.psClusterDefinition = &sCLD_SMDummy;
#endif

#ifdef TIME_CLIENT
```

```
        /* Create Time client. Share the cluster data from endpoint1 */
        if (eCLD_TimeCreateTime(
                     &sIPDDevice.sClusterInstance.sTimeClient,
                     TRUE,
                     &sCLD_Time,
                     &sSE_IPDDevice.sTimeCluster,
                     &au8AppTimeClientAttributeControlBits[0]
                     ) != E_ZCL_SUCCESS)
        {
            return E_ZCL_FAIL;
        }
#else
      sIPDDevice .sClusterInstance. sTimeClient.psClusterDefinition = &sCLD_TimeDummy;
#endif

#ifdef DRLC_CLIENT
        /* Create DRLC client. Use different cluster data */
        if (eSE_DRLCCreate(
            FALSE,
            SE_DRLC_NUMBER_OF_CLIENT_LOAD_CONTROL_ENTRIES,
           &au8AppDRLCClientAttributeControlBits[0],
           &sIPDDevice.sClusterInstance.sDRLCClient,
           &sCLD_DRLC,
           &sIPDDevice.sDRLCCustomDataStructure,
           sIPDDevice.asDRLCLoadControlEventRecord,
           &sIPDDevice.sDRLCCluster   )
            != E_ZCL_SUCCESS)
        {
            return E_ZCL_FAIL;
        }
#else
      sIPDDevice .sClusterInstance. sDRLCClient.psClusterDefinition = &sCLD_DRLCDummy;
#endif

#ifdef CLD_KEY_ESTABLISHMENT
        /* Create KEC client. Share the cluster data from endpoint1 */
        if (eSE_KECCreate(
&au8AppKECClientAttributeControlBits[0],
            & sIPDDevice.sClusterInstance.sKeyEstablishmentClient,
            &sCLD_KeyEstablishment,
            &sSE_IPDDevice.sKECCustomDataStructure, /* Information shared with endpoint1 */
            &sSE_IPDDevice.sKeyEstablishmentCluster,) /* Information shared with endpoint1 */
            != E_ZCL_SUCCESS)

        {
            return E_ZCL_FAIL;
        }
#else
      sIPDDevice .sClusterInstance. sKeyEstablishmentCluster.psClusterDefinition = &sCLD_KECDummy;
#endif
#ifdef MC_CLIENT
        /* Create Messaging client. Share the cluster data from endpoint1 */
        if (eSE_MCCreate(
            FALSE,
            SE_MESSAGE_NUMBER_OF_CLIENT_MESSAGE_RECORD_ENTRIES,
            &sSE_IPDDevice.au8DisplayMessage[0][0], /* Information shared with endpoint1 */
            &sIPDDevice.sClusterInstance.sMCClient,
            &sCLD_MC,
            &sSE_IPDDevice.sMessageCustomDataStructure, /* Information shared with endpoint1 */
            sSE_IPDDevice.asDisplayMessageCommandPayloadRecord) /* Information shared with endpoint1 */
            != E_ZCL_SUCCESS)

        {
                        return E_ZCL_FAIL;
        }
#else
      sIPDDevice .sClusterInstance. sMCClient.psClusterDefinition = &sCLD_MCDummy;
#endif

        /* Create Identify client. Share the cluster data from endpoint1 */
       if(eCLD_IdentifyCreateIdentify(
                     &sIPDDevice.sClusterInstance.sIdentifyClient,
                     FALSE,
                     &sCLD_Identify,
                     &sSE_IPDDevice.sIdentifyClientCluster,
                     (void*)&au8AppIdentifyServerAttributeControlBits[0],
                     &sSE_IPDDevice.sIdentifyClientCustomDataStructure
                     ) != E_ZCL_SUCCESS)
       {
           return E_ZCL_FAIL;
       }
        /* Create Price client. Use different cluster data from endpoint1 */
#ifdef PRICE_CLIENT
        if (eSE_PriceCreate(
          FALSE,
          SE_PRICE_NUMBER_OF_CLIENT_PRICE_RECORD_ENTRIES,
          &au8AppPriceClientAttributeControlBits[0],
          &sIPDDevice.au8RateLabel[0][0]
          &sIPDDevice.sClusterInstance.sPriceClient,
          &sCLD_Price,
          &sIPDDevice.sPriceCustomDataStructure,
          sIPDDevice.asPublishPriceRecord,
          &sIPDDevice.sPriceCluster)

        ) != E_ZCL_SUCCESS)
        {
                    return E_ZCL_FAIL;
        }
#else
      sIPDDevice .sClusterInstance. sPriceClient.psClusterDefinition = &sCLD_PriceDummy;
#endif

        eZCL_Status =  eZCL_Register(&sIPDDevice.sEndPoint);
        if(E_ZCL_SUCCESS != eZCL_Status)
        {
            return eZCL_Status;
        }

    return E_ZCL_SUCCESS;
}
```

Revision History

Version	Date	Comments
1.0	03-Dec-2009	First release
1.1	22-Mar-2010	Minor updates
2.0	24-Nov-2010	Incorporated information from former *ZigBee PRO Smart Energy API Reference Manual (JN-RM-2046)*
3.0	10-May-2011	Details of clusters from the ZigBee Cluster Library (ZCL) migrated to the *ZCL User Guide (JN-UG-3077)*. Key Establishment testing added and OTA Upgrade cluster introduced
3.1	23-May-2012	Made minor updates/corrections and added: • Demand-Response and Load Control (DRLC) cluster • Extra features (Mirroring, Get Profile) to Simple Metering cluster
3.2	24-Aug-2012	Updated to include ‘calorific value’ and ‘conversion factor’ in the Price cluster. Other minor updates/corrections also made
3.3	14-Jan-2013	Updated various structure definitions and Price cluster attribute sets
3.4	22-Apr-2013	Made minor updates/corrections and added: • Tunnelling cluster • Custom endpoints

Important Notice

NXP Laboratories UK Ltd
(Formerly Jennic Ltd)
Furnival Street
Sheffield
S1 4QT
United Kingdom

Tel: +44 (0)114 281 2655
Fax: +44 (0)114 281 2951

For the contact details of your local NXP office or distributor, refer to:

www.nxp.com/jennic