JNUG3132

ZigBee Cluster Library (for ZigBee 3.0)

Rev. 3.1 — 24 January 2025

Document information

Information	Content
Keywords	JNUG3132, NXP K32W041, K32W061, K32W1, MCXW71, MCXW72, and JN518x family of wireless microcontrollers, ZigBee devices, clusters, attributes, ZigBee 3.0 standard, ZigBee 3.0 applications, ZigBee 3.0 Software Development Kit (SDK)
Abstract	This manual describes the NXP implementation of the ZigBee Cluster Library (ZCL) for the ZigBee 3.0 standard on the K32W041, K32W061, K32W1, MCXW71, MCXW72 and JN518x family of wireless microcontrollers. The NXP hardware platforms: K32W148-EVK, FRDM-MCXW71, FRDM-MCXW72, MCX-W71-EVK, and MCX-W72-EVK support this library.

Preface

Introduction

This manual describes the NXP implementation of the ZigBee Cluster Library (ZCL) for the ZigBee 3.0 standard. The manual describes the clusters from the ZCL that may be used in ZigBee 3.0 applications developed using an NXP ZigBee 3.0 Software Developer's Kit (SDK).

Prerequisites

This manual assumes that you are already familiar with the concepts of ZigBee devices, clusters, and attributes. These are described in the ZigBee 3.0 Stack User Guide (JNUG3130), available from the NXP web site (see "Support Resources").

Chip Compatibility

The ZCL software described in this manual can be used on the NXP K32W041, K32W061, K32W1, MCXW71, MCXW72, and JN518x family of wireless microcontrollers. The NXP hardware platforms K32W148-EVK, FRDM-MCXW71, FRDM-MCXW72, MCX-W71-EVK, and MCX-W72-EVK are supported.

Organization

- Part I: Fundamentals comprises four chapters:
  - Chapter 1 introduces the ZigBee Cluster Library (ZCL)
  - Chapter 2 describes some essential concepts for the ZCL, including read/write access to cluster attributes and the associated read/write functions
  - Chapter 3 describes the event handling framework of the ZCL, including the supplied event handling function
  - Chapter 4 describes the error handling provision of the ZCL, including the supplied error handling function
- Part II: Common Resources comprises three chapters:
  - Chapter 5 details the general functions of the ZCL
  - Chapter 6 details the general structures used by the ZCL
  - Chapter 7 details the general enumerations used by the ZCL
- Part III: General Clusters comprises fifteen chapters:
  - Chapter 8 details the **Basic** cluster
  - Chapter 9 details the **Power Configuration** cluster
  - Chapter 10 details the **Device Temperature Configuration** cluster
  - Chapter 11 details the **Identify** cluster
  - Chapter 12 details the **Groups** cluster
  - Chapter 13 details the **Scenes** cluster
  - Chapter 14 details the **On/Off** cluster
  - Chapter 15 details the **On/Off Switch Configuration** cluster
  - Chapter 16 details the **Level Control** cluster
  - Chapter 17 details the **Alarms** cluster
  - Chapter 18 details the **Time** cluster, as well as the use of ZCL time
  - Chapter 19 details the **Input and Output** clusters
  - Chapter 20 details the **Poll Control** cluster
  - Chapter 21 details the **Power Profile** cluster

  - Chapter 22 details the **Diagnostics** cluster
- Part IV: Measurement and Sensing Clusters comprises eight chapters:
  - Chapter 23 details the **Illuminance Measurement** cluster
  - Chapter 24 details the **Illuminance Level Sensing** cluster
  - Chapter 25 details the **Temperature Measurement** cluster
  - Chapter 26 details the **Pressure Measurement** cluster
  - Chapter 27 details the **Flow Measurement** cluster
  - Chapter 28 details the **Relative Humidity Measurement** cluster
  - Chapter 29 details the **Occupancy Sensing** cluster
  - Chapter 30 details the **Electrical Measurement** cluster
- Part V: Lighting Clusters comprises two chapters:
  - Chapter 31 details the **Colour Control** cluster
  - Chapter 32 details the **Ballast Configuration** cluster
- Part VI: HVAC Clusters comprises three chapters:
  - Chapter 33 details the **Thermostat** cluster
  - Chapter 34 details the **Fan Control** cluster
  - Chapter 35 details the **Thermostat UI Configuration** cluster
- Part VII: Closure Clusters comprises one chapter:
  - Chapter 36 details the **Door Lock** cluster
- Part VIII: Security and Safety Clusters comprises three chapters:
  - Chapter 37 details the **IAS Zone** cluster
  - Chapter 38 details the **IAS ACE (Ancillary Control Equipment)** cluster
  - Chapter 39 details the **IAS WD (Warning Device)** cluster
- Part IX: Smart Energy Clusters comprises three chapters:
  - Chapter 40 details the **Price** cluster
  - Chapter 41 details the **Demand-Response and Load Control** cluster
  - Chapter 42 details the **Simple Metering** cluster
- Part X: Commissioning Clusters comprises two chapters:
  - Chapter 43 details the **Commissioning** cluster
  - Chapter 44 details the **Touchlink Commissioning** cluster
- Part XI: Appliances Clusters comprises four chapters:
  - Chapter 45 details the **Appliance Control** cluster
  - Chapter 46 details the **Appliance Identification** cluster
  - Chapter 47 details the **Appliance Events and Alerts** cluster
  - Chapter 48 details the **Appliance Statistics** cluster
- Part XII: Over-The-Air Upgrade comprises one chapter:
  - Chapter 49 details the **OTA (Over-the-Air) Upgrade** cluster
- Part XIII: Appendices comprises the nine appendixes listed below:
  - Appendix A: Mutex Callbacks
  - Appendix B: Attribute Reporting
  - Appendix C: Extended Attribute Discovery
  - Appendix D: Custom Endpoints
  - Appendix E: Manufacturer-specific Attributes and Commands
  - Appendix F: OTA Extension for Dual-processor Nodes
  - Appendix G: Glossary

    These cover topics that include mutex callbacks, attribute reporting, attribute discovery, custom endpoints, manufacturer-specific attributes and commands. The storage of OTA upgrade applications in internal or

external flash memory, OTA upgrade of nodes comprising two processors, example code fragments, and a glossary of terms are also included.

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.

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

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

Acronyms

Table 1. Acronyms

S.No	Acronym	Description
1	ACE	Ancillary Control Equipment
2	API	Application Programming Interface
3	APDU	Application Protocol Data Unit
4	CIE	Control and Indicating Equipment
5	DRLC	Demand-Response and Load Control
6	HVAC	Heating, Ventilation and Air-Conditioning
7	IAS	Intruder Alarm System
8	OTA	Over The Air
9	SE	Smart Energy
10	UI	User Interface
11	UTC	Co-ordinated Universal Time
12	WD	Warning Device
13	ZCL	ZigBee Cluster Library

Related Documents

Refer to the following documents for further information:

- JNUG3130 ZigBee 3.0 Stack User Guide
- JNUG3131 ZigBee Devices User Guide
- Connectivity Framework Reference Manual
- 075123 rev 7 ZigBee Cluster Library Specification [from ZigBee Alliance]
- 095264 ZigBee Over-the-air Upgrading Cluster [from ZigBee Alliance]

Support Resources

To access online support resources such as SDKs, Application Notes and User Guides, visit the Wireless Connectivity page of the NXP web site:

- www.nxp.com/products/wireless-connectivity

All NXP resources referred to in this manual can be found at the above address, unless otherwise stated.

Part I: Fundamentals

Part 1 comprises the following:

- Chapter 1 introduces the ZigBee Cluster Library (ZCL)
- Chapter 2 describes some essential concepts for the ZCL, including read/write access to cluster attributes and the associated read/write functions
- Chapter 3 describes the event handling framework of the ZCL, including the supplied event handling function
- Chapter 4 describes the error handling provision of the ZCL, including the supplied error handling function

1 ZigBee Cluster Library (ZCL)

The ZigBee Cluster Library (ZCL) for ZigBee 3.0 contains standard clusters, as defined by the ZigBee Alliance, for use in ZigBee 3.0 applications over a diverse range of market sectors. Each cluster corresponds to a specific functionality, through a set of attributes and/or commands. Clusters can be selected from the ZCL to give an application the required set of capabilities.

The ZCL also 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 integers and strings), common commands (for example, for reading attributes), and default responses for indicating success or failure.

The NXP implementation of the ZCL, described in this manual, is supplied in the ZigBee 3.0 Software Developer's Kit (SDK) available via the NXP web site (see Section "Support Resources"). These SDKs provide only the clusters supported by NXP that are described in this manual. The ZCL is fully detailed in the *ZigBee Cluster Library Specification (075123)*, available from the ZigBee Alliance.

1.1 ZCL Member Clusters

The clusters of the ZCL are divided into functional areas, for convenience. An application can use clusters from any number of these areas to make up its complete functionality. The clusters implemented by NXP are from the following areas (the associated clusters are listed in the referenced sub-sections):

- General - Section 1.1.1
- Measurement and Sensing - Section 1.1.2
- Lighting - Section 1.1.3
- Heating, Ventilation and Air-Conditioning (HVAC) - Section 1.1.4
- Closures - Section 1.1.5
- Security and Safety - Section 1.1.6
- Smart Energy - Section 1.1.7
- Commissioning - Section 1.1.8
- Appliances - Section 1.1.9
- Over-The-Air (OTA) Upgrade - Section 1.1.10

Note: *Not all of the clusters from the above ZCL functional areas are available in the NXP software.*

1.1.1 General

The General clusters implemented by NXP are listed and outlined in the table below. These clusters are detailed in 'Part III: General Clusters' of this manual.

Table 2. General Clusters

Cluster	Cluster ID	Description
Basic	0x0000	The Basic cluster contains the basic properties of a ZigBee device for example, software and hardware versions. The Basic cluster allows the setting of user-defined properties such as location. This cluster is detailed in Chapter 8.
Power Configuration	0x0001	The Power Configuration cluster allows users to determine the power source details of a device and helps configure the under/over-voltage alarms. This cluster is detailed in Chapter 9.
Device Temperature Configuration	0x0002	The Device Temperature Configuration cluster allows information about the internal temperature of a device to be obtained and under/over-temperature alarms to be configured. This cluster is detailed in Chapter 10.

Table 2. General Clusters*...continued*

Cluster	Cluster ID	Description
Identify	0x0003	The Identify cluster allows a ZigBee device to make itself known visually (for example, by flashing a light) to an observer, such as a network installer. This cluster is detailed in Chapter 11.
Groups	0x0004	The Groups cluster allows the management of the Group table concerned with group addressing - that is, the targeting of multiple endpoints using a single address. This cluster is detailed in Chapter 12.
Scenes	0x0005	The Scenes cluster allows the management of pre-defined sets of cluster attribute values called scenes, where a scene can be stored, retrieved, and applied to put the system into a pre-determined state. This cluster is detailed in Chapter 13.
On/Off	0x0006	The On/Off cluster allows a device to be put into the ‘on’ and ‘off’ states, or toggled between the two states. This cluster is detailed in Chapter 14.
On/Off Switch Configuration	0x0007	The On/Off Switch Configuration cluster allows the switch type on a device to be defined, as well as the commands to be generated when the switch is moved between its two states. This cluster is detailed in Chapter 15.
Level Control	0x0008	The Level Control cluster allows control of the level of a physical quantity (for example, heat output) on a device. This cluster is detailed in Chapter 16.
Alarms	0x0009	The Alarms cluster is used for sending alarm notifications and the general configuration of alarms for all other clusters on the ZigBee device (individual alarm conditions are set in the corresponding clusters). This cluster is detailed in Chapter 17.
Time	0x000A	The Time cluster provides an interface to a real-time clock on a ZigBee device, allowing the clock time to be read and written in order to synchronize the clock to a time standard. This is the number of seconds since 0 hrs 0 mins 0 secs on 1st January 2000 UTC (Coordinated Universal Time). This cluster includes functionality for local time-zone and daylight saving time. This cluster is detailed in Chapter 18.
Analogue Input (Basic)	0x000C	The Analogue Input (Basic) cluster provides an interface for accessing an analog measurement. This cluster is detailed in Section 19.1.
Analogue Output (Basic)	0x000D	The Analogue Output (Basic) cluster provides an interface for setting the value of an analog output. This cluster is detailed in Section 19.2.
Binary Input (Basic)	0x000F	The Binary Input (Basic) cluster provides an interface for accessing a binary (two-state) measurement. This cluster is detailed in Section 19.3.
Binary Output (Basic)	0x0010	The Binary Output (Basic) cluster provides an inter-face for setting the state of a binary (two-state) output. This cluster is detailed in Section 19.4.
Multistate Input (Basic)	0x0012	The Multistate Input (Basic) cluster provides an interface for accessing a multistate measurement (that can take one of a set of fixed states). This cluster is detailed in Section 19.5.
Multistate Output (Basic)	0x0013	The Multistate Output (Basic) cluster provides an interface for setting the value of a multistate output (that can take one of a set of fixed states). This cluster is detailed in Section 19.6.
Poll Control	0x0020	The Poll Control cluster provides an interface for remotely controlling the rate at which a ZigBee End Device polls its parent for data. This cluster is detailed in Chapter 20.
Power Profile	0x001A	The Power Profile cluster provides an interface between a home appliance (for example, a washing machine) and the controller of an energy management system. This cluster is detailed in Chapter 21.

Table 2. General Clusters*...continued*

Cluster	Cluster ID	Description
Diagnostics	0x0B05	The Diagnostics cluster allows the operation of the ZigBee PRO stack to be followed over time. This cluster is detailed in Chapter 22.

1.1.2 Measurement and Sensing

The Measurement and Sensing clusters implemented by NXP are listed and outlined in the table below. These clusters are detailed in 'Part IV: Measurement and Sensing Clusters' of this manual.

Table 3. Measurement and Sensing Clusters

Cluster	Cluster ID	Description
Illuminance Measurement	0x0400	The Illuminance Measurement cluster provides an interface to an illuminance measuring device, allowing the configuration of measuring and the reporting of measurements. This cluster is detailed in Chapter 23.
Illuminance Level Sensing	0x0401	The Illuminance Level Sensing cluster provides an interface to light-level sensing functionality. This cluster is detailed in Chapter 24.
Temperature Measurement	0x0402	The Temperature Measurement cluster provides an interface to a temperature measuring device, allowing the configuration of measuring and the reporting of measurements. This cluster is detailed in Chapter 25.
Pressure Measurement	0x0403	The Pressure Measurement cluster provides an inter-face to a pressure measuring device, allowing the configuration of measuring and the reporting of measurements. This cluster is detailed in Chapter 26.
Flow Measurement	0x0404	The Flow Measurement cluster provides an interface to a flow measuring device for a fluid, allowing the configuration of measuring and the reporting of measurements. This cluster is detailed in Chapter 27.
Relative Humidity Measurement	0x0405	The Relative Humidity Measurement cluster provides an interface to a humidity measuring device, allowing the configuration of relative humidity measuring and the reporting of measurements. This cluster is detailed in Chapter 28.
Occupancy Sensing	0x0406	The Occupancy Sensing cluster provides an interface to an occupancy sensor, allowing the configuration of sensing and the reporting of status. This cluster is detailed in Chapter 29.
Electrical Measurement	0x0B04	The Electrical Measurement cluster provides an interface for obtaining electrical measurements from a device. This cluster is detailed in Chapter 30.

1.1.3 Lighting

The Lighting clusters implemented by NXP are listed and outlined in the table below. These clusters are detailed in 'Part V: Lighting Clusters' of this manual.

Table 4. Lighting Clusters

Cluster	Cluster ID	Description
Colour Control	0x0300	The Colour Control cluster can be used to adjust the colour of a light (it does not govern the overall luminance of the light, as this is controlled using the Level Control cluster). This cluster is detailed in Chapter 31.

Table 4. Lighting Clusters*...continued*

Cluster	Cluster ID	Description
Ballast Configuration	0x0301	The Ballast Configuration cluster can be used to configure a lighting ballast that restricts the light levels of a connected set of lamps. This cluster is detailed in Chapter 32.

1.1.4 Heating, Ventilation, and Air-Conditioning (HVAC)

The HVAC clusters implemented by NXP are listed and outlined in the table below. These clusters are detailed in 'Part VI: HVAC Clusters' of this manual.

Table 5. HVAC Clusters

Cluster	Cluster ID	Description
Thermostat	0x0201	The Thermostat cluster provides a means of configuring and controlling the functionality of a thermostat. This cluster is detailed in Chapter 33.
Fan Control	0x0202	The Fan Control cluster provides a means of con-trolling the speed or state of a fan which may be part of a heating or cooling system. The cluster is detailed in Chapter 34.
Thermostat User Interface Configuration	0x0204	The Thermostat User Interface (UI) Configuration cluster provides a means of configuring the user inter-face (keypad and/or LCD screen) for a thermostat or a thermostat controller device. This cluster is detailed in Chapter 35.

1.1.5 Closures

The Closure clusters implemented by NXP are listed and outlined in the table below. These clusters are detailed in 'Part VII: Closure Clusters' of this manual.

Table 6. Closure Clusters

Cluster	Cluster ID	Description
Door Lock	0x0101	The Door Lock cluster provides a means of representing the state of a door lock and (optionally) the door. This cluster is detailed in Chapter 36.

1.1.6 Security and Safety

The Security and Safety clusters implemented by NXP are listed and outlined in the table below. These clusters are detailed in 'Part VIII: Security and Safety Clusters' of this manual.

Table 7. Security and Safety Clusters

Cluster	Cluster ID	Description
IAS Zone	0x0500	The IAS Zone cluster provides an interface to a zone device in an IAS (Intruder Alarm System). This cluster is detailed in Chapter 37.
IAS ACE (Ancillary Control Equipment)	0x0501	The IAS ACE cluster provides a control interface to a CIE (Control and Indicating Equipment) device in an IAS (Intruder Alarm System). This cluster is detailed in Chapter 38.
IAS WD (Warning Device)	0x0502	The IAS WD cluster provides an interface to a Warning Device in an IAS (Intruder Alarm System). For example, a CIE (Control and Indicating Equipment) device can use the cluster to issue alarm warning indications to a Warning Device when an alarm condition is detected. This cluster is detailed in Chapter 39.

1.1.7 Smart Energy

The Smart Energy clusters implemented by NXP are listed and outlined in the table below. These clusters are detailed in 'Part IX: Smart Energy Clusters' of this manual.

Table 8. Smart Energy Clusters

Cluster	Cluster ID	Description
Price	0x0700	The Price cluster provides the mechanism for sending and receiving pricing information within a ZigBee 3.0 network. This cluster is detailed in Chapter 40.
Demand-Response and Load Control	0x0701	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 and act upon them - the demand-response functionality. This cluster is detailed in Chapter 41.
Simple Metering	0x0702	The Simple Metering cluster provides a mechanism to obtain consumption data from a metering device (elec-tric, gas, water or thermal). This cluster is detailed in Chapter 42.

Other Smart Energy (SE) clusters, that are not available in the ZigBee 3.0 SDK, are provided in the NXP ZigBee Smart Energy SDK. The ZigBee 3.0 SDK contains only the SE clusters that are supported by NXP for non-SE applications.

1.1.8 Commissioning

The Commissioning clusters implemented by NXP are listed and outlined in the table below. These clusters are detailed in 'Part X: Commissioning Clusters' of this manual.

Table 9. Commissioning Clusters

Cluster	Cluster ID	Description
Commissioning	0x0015	The Commissioning cluster can be 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). This cluster is detailed in Chapter 43.
Touchlink Commissioning	0x1000	The Touchlink Commissioning cluster is used when forming a ZigBee 3.0 network or adding a new node to an existing network. This cluster is detailed in Chapter 44.

1.1.9 Appliances

The Appliances clusters implemented by NXP are listed and outlined in the table below. These clusters are detailed in 'Part XI: Appliances Clusters' of this manual.

Table 10. Appliances Clusters

Cluster	Cluster ID	Description
Appliance Control	0x001B	The Appliance Control cluster provides an interface for remotely controlling appliances in the home. This cluster is detailed in Chapter 45.
Appliance Identification	0x0B00	The Appliance Identification cluster provides an interface for obtaining and setting basic appliance information. This cluster is detailed in Chapter 46.

Table 10. Appliances Clusters*...continued*

Cluster	Cluster ID	Description
Appliance Events and Alerts	0x0B02	The Appliance Events and Alerts cluster provides an interface for the notification of significant events and alert situations. This cluster is detailed in Chapter 47.
Appliance Statistics	0x0B03	The Appliance Statistics cluster provides an interface for supplying statistical information about an appliance. This cluster is detailed in Chapter 48.

1.1.10 Over-The-Air (OTA) Upgrade

The Over-The-Air (OTA) Upgrade cluster is outlined in the table below and detailed in 'Part XII: Over-The-Air Upgrade' of this manual.

Table 11. OTA Upgrade Cluster

Cluster	Cluster ID	Description
OTA Upgrade	0x0019	The OTA Upgrade cluster provides the facility to upgrade (or downgrade or re-install) application software on the nodes of a ZigBee 3.0 network by distributing the replacement software through the network (over the air) and updating the software with minimal interruption to node operation. This cluster is detailed in Chapter 49.

1.2 General ZCL Resources

In addition to clusters, the ZCL provides general (non-cluster-specific) resources. For example, common mechanisms are used to allow a cluster client to access (read and write to) the attributes on the cluster server - the NXP ZCL software includes C functions and structures for performing such accesses across all clusters.

The fundamental principles and mechanisms of the ZCL are presented in the rest of Part I: Fundamentals:

- Chapter 2 describes essential ZCL principles, such as accessing attributes
- Chapter 3 describes the ZCL handling of stack-related and timer-related events
- Chapter 4 described the ZCL handling of errors.

The general resources provided by the ZCL software are detailed in Part II: Common Resources:

- Chapter 5 details the core functions provided with the ZCL
- Chapter 6 details the general ZCL structures
- Chapter 7 details the general ZCL enumerations and status codes

Cluster-specific resources are detailed in the respective chapters for the clusters.

Note: *It is possible to customize the clusters of the ZCL by introducing manufacturer-specific attributes and commands. The processes of adding custom attributes and commands are described in Appendix E.*

1.3 ZCL Compile-time Options

Before the application can be built, the ZCL compile-time options must be configured in the header file **zcl_options.h** for the application.

Enabled Clusters

All required clusters must be enabled in the options header file. For example, to enable the Basic and Time clusters, the following lines are required:

```
#define CLD_BASIC
#define CLD_TIME
```

In addition, to include the software for a cluster client or server or both, it is necessary to select them in the options header file. For example, to select the Basic cluster client and server, the following lines are required:

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

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
```

Each of the above definitions apply to all clusters used in the application.

Tip: *If only read access to attributes is required, then it is recommended not to enable write access. This is because omitting the write options gives the benefit of a reduced application size.*

Optional Attributes

Many clusters have optional attributes that may be enabled at compile-time via the options header file - for example, to enable the Time Zone attribute in the Time cluster:

```
#define E_CLD_TIME_ATTR_TIME_ZONE
```

Note: *Cluster-specific compile-time options are described in detail in the chapters for the individual clusters. The attribute reporting feature also has its own compile-time options (see Appendix B.3.1).*

Number of Endpoints

The following line can be added to set the number of endpoints supported on a node (to n):

```
#define ZCL_NUMBER_OF_ENDPOINTS   n
```

The default value is 1.

Default Responses

The following line can be added to enable default responses (see Section 2.5):

```
#define ZCL_DISABLE_DEFAULT_RESPONSES
```

By default, they are disabled.

APS Acknowledgements

The following line can be added to enable APS acknowledgments:

```
#define ZCL_DISABLE_APS_ACK
```

By default, they are disabled.

Cooperative Tasks

By default, tasks within the application are cooperative, and events will therefore not be generated for locking and unlocking mutexes for resources that are shared between the tasks. To disable the cooperative task feature, add the following line:

```
#undefine COOPERATIVE
```

Note that this requires an undefine.

Parameter Checking

Parameter checking in various functions can be enabled by including the following line:

```
#define STRICT_PARAM_CHECK
```

This feature is useful for testing during application development. When the testing is complete, the option should be disabled to eliminate the checks and to save code memory. This option can be defined in the **zcl_options.h** file or the makefile.

‘Wild Card’ Profile

Commands with a ‘wild card’ application profile (Profile ID of 0xFFFF) can be accepted and processed by the receiving device by including the following line:

```
#define ZCL_ALLOW_WILD_CARD_PROFILE
```

2 ZCL Fundamentals and Features

This chapter describes essential ZCL concepts, including the use of shared device structures as well as remote read and write accesses to cluster attributes.

Note:

1. *This chapter assumes that you are familiar with ZigBee clusters and associated concepts (such as the cluster server and client). For an introduction to ZigBee clusters, refer to the ZigBee 3.0 Stack User Guide (JNUG3130).*
2. *The ZCL functions referred to in this chapter are detailed in Chapter 5.*

2.1 Initializing the ZCL

The ZCL can be initialized using the function **eZCL_Initialise()**, which must be called before registering any endpoints. The initialization is done using the device-specific endpoint registration functions and before starting the ZigBee PRO stack. As part of this initialization, you must specify a user-defined callback function that would be invoked when a ZigBee PRO stack event occurs that is not associated with an endpoint. Also provide a local pool of Application Protocol Data Units (APDUs) that are used by the ZCL to hold messages that are to be sent and received.

2.2 Shared Device Structures

In each ZigBee device, cluster attribute values are exchanged between the application and the ZCL by means of a shared structure. This structure is protected by a mutex - see Appendix A. The structure for a particular ZigBee device contains structures for the clusters supported by that device.

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 1.3.*

A shared device structure within a device can be accessed both by the local application and by a remote application on another device. Remote read and write operations involving a shared device structure are illustrated in Figure 1 below. Normally, a cluster client requests these operations and they are performed on a cluster server. For more detailed descriptions of these operations, refer to Section 2.3.

Usually, the ZCL parses remote commands that write attribute values to the shared device structure. The written values can then be read by the local application. For example, an On/Off Switch device remotely writes to the shared device structure in an On/Off Light device and the local application then reads this data to change the state or configuration of the light.

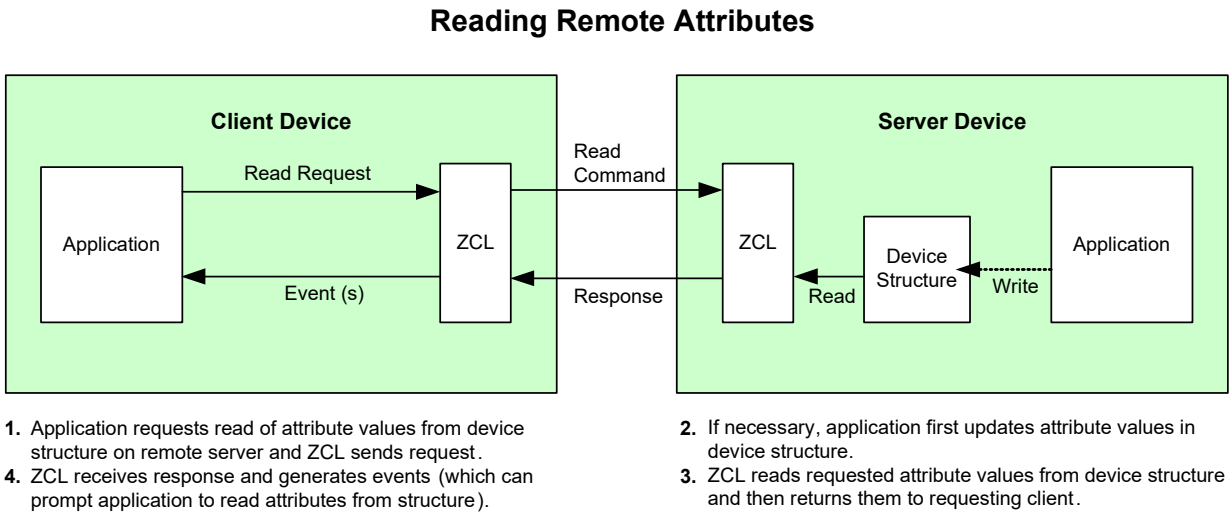


Figure 1. Operations using Shared Device Structure

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).*

2.3 Accessing Attributes

This section describes the processes of reading and writing cluster attributes on a remote node. For the attribute access function descriptions, refer to Section 5.2.

2.3.1 Attribute Access Permissions

For each attribute of a cluster, access permissions should be defined for the different types of access to the attribute. These permissions are configured using control flags, with one flag for each access type, as follows:

Table 12. Attribute Access Types and Control Flags

Access Type	Flag	Description
Read	E_ZCL_AF_RD	Global commands can read the attribute value
Write	E_ZCL_AF_WR	Global commands can write a new value to the attribute
Report	E_ZCL_AF_RP	Global commands can report the value of the attribute or configure the attribute for default reporting
Scene	E_ZCL_AF_SE	The attribute can be accessed through a scene (if the Scenes cluster is implemented on the same endpoint)

If a particular access type is required for an individual attribute, the corresponding flag must be defined for that attribute. This is done in the C header file for the cluster. For example, in the case of the On/Off cluster, the required flags must be defined for each attribute in the following structure in the **OnOff.c** file:

```
const tsZCL_AttributeDefinition asCLD_OnOffClusterAttributeDefinitions[] = {
#ifdef ONOFF_SERVER
  {E_CLD_ONOFF_ATTR_ID_ONOFF, (E_ZCL_AF_RD|E_ZCL_AF_SE|E_ZCL_AF_RP), E_ZCL_BOOL,
  (uint32)(&((tsCLD_OnOff*)(0))->bOnOff),0}, /* Mandatory */
#ifdef CLD_ONOFF_ATTR_GLOBAL_SCENE_CONTROL
  {E_CLD_ONOFF_ATTR_ID_GLOBAL_SCENE_CONTROL, (E_ZCL_AF_RD), _ZCL_BOOL,
  (uint32)(&((tsCLD_OnOff*)(0))->bGlobalSceneControl),0}, /* Optional */
#endif
#ifdef CLD_ONOFF_ATTR_ON_TIME
  {E_CLD_ONOFF_ATTR_ID_ON_TIME, E_ZCL_AF_RD|E_ZCL_AF_WR), E_ZCL_UINT16,
  (uint32)(&((tsCLD_OnOff*)(0))->u16OnTime),0}, /* Optional */
#endif
#ifdef CLD_ONOFF_ATTR_OFF_WAIT_TIME
  {E_CLD_ONOFF_ATTR_ID_OFF_WAIT_TIME, (E_ZCL_AF_RD|E_ZCL_AF_WR), E_ZCL_UINT16,
  (uint32)(&((tsCLD_OnOff*)(0))->u16OffWaitTime),0}, /* Optional */
#endif
#ifdef CLD_ONOFF_ATTR_STARTUP_ONOFF
  /* ZLO extension for OnOff Cluster */
  {E_CLD_ONOFF_ATTR_ID_STARTUP_ONOFF, E_ZCL_AF_RD|E_ZCL_AF_WR), E_ZCL_ENUM8,
  (uint32)(&((tsCLD_OnOff*)(0))->eStartUpOnOff),0}, /* Optional */
#endif
#endif
  {E_CLD_GLOBAL_ATTR_ID_CLUSTER_REVISION, (E_ZCL_AF_RD|E_ZCL_AF_GA), E_ZCL_UINT16,
  (uint32)(&((tsCLD_OnOff*)(0))->u16ClusterRevision),0}, /* Mandatory */
  #if (defined ONOFF_SERVER) && (defined CLD_ONOFF_ATTR_ATTRIBUTE_REPORTING_STATUS)
  {E_CLD_GLOBAL_ATTR_ID_ATTRIBUTE_REPORTING_STATUS,(E_ZCL_AF_RD|E_ZCL_AF_GA), E_ZCL_ENUM8,
  (uint32)(&((tsCLD_OnOff*)(0))->u8AttributeReportingStatus),0}, /* Optional */
#endif
};
```

Note: *The flag E_ZCL_AF_GA indicates a global attribute.*

2.3.2 Reading Attributes

A ZigBee 3.0 application might require to read attribute values from a remote device. Attributes are read by sending a ‘read attributes’ request, normally from a client cluster to a server cluster. This request can be sent using a general ZCL function (see below) or using a 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: *Users should enable read access to cluster attributes explicitly at compile-time as described in Section 1.3.*

A ZCL function is provided for reading a set of attributes of a remote cluster instance, as described in Section 2.3.2.1. A function is also provided for reading a local cluster attribute value, as described in Section 2.3.2.2.

2.3.2.1 Reading a set of attributes of a remote cluster

This section describes the use of the function **eZCL_SendReadAttributesRequest()** to send a 'read attributes' request to a remote cluster in order to obtain the values of selected attributes. The resulting activities on the source and destination nodes are outlined below and illustrated in Figure 2. The events generated from a 'read attributes' request are further described in Chapter 3.

Note: *The described sequence is similar when using the cluster-specific 'read attributes' functions.*

1. On Source Node

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

On receiving the 'read attributes' request, the ZCL 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. If tasks within the application are not cooperative, the ZCL 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 Appendix A.

3. Reads the relevant attribute values from the shared device structure and creates a 'read attributes' response message containing the read values.

4. If tasks within the application are not cooperative, the ZCL 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

On receiving the 'read attributes' response, the ZCL software on the source node performs the following steps:

1. 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.

2. 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.

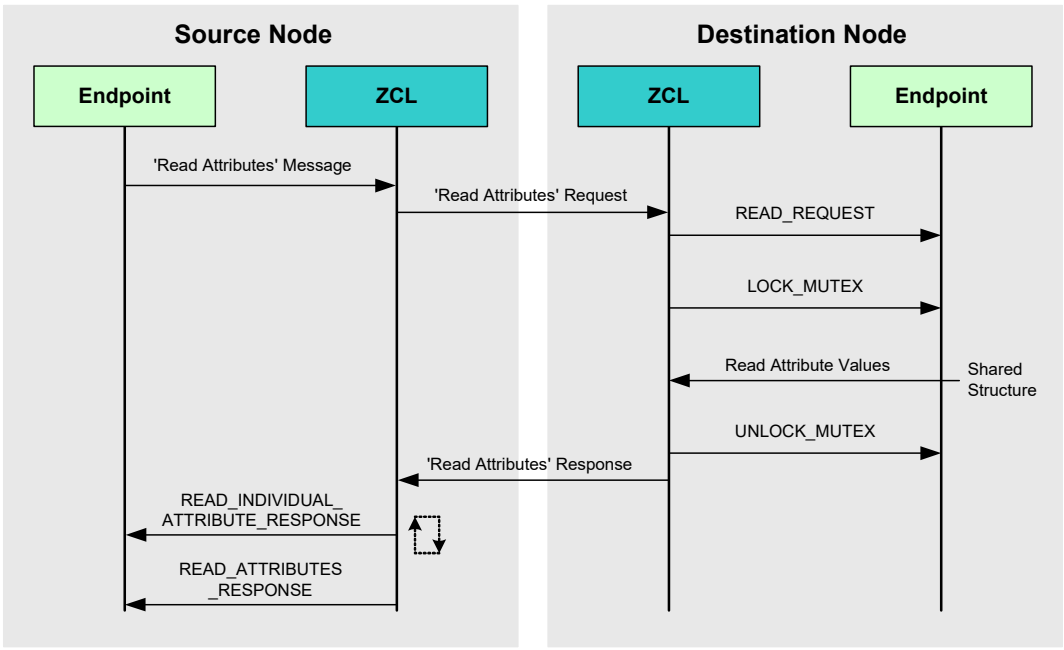


Figure 2. 'Read Attributes' Request and Response

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 3.2.*

2.3.2.2 Reading an Attribute of a Local Cluster

An individual attribute of a cluster on the local node can be read using the function **eZCL_ReadLocalAttributeValue()**. The read value is returned by the function (in a memory location for which a pointer must be provided).

2.3.3 Writing Attributes

The ZCL provides functions for writing attribute values to both remote and local clusters, as described in Section 2.3.3.1 and Section 2.3.3.2 respectively.

2.3.3.1 Writing to Attributes of a Remote Cluster

A ZigBee 3.0 application might require to write attribute values to a remote device. Attribute values are written by sending a 'write attributes' request, normally from a client cluster to a server cluster, where the relevant attributes in the shared device structure are updated. Write access to cluster attributes must be explicitly enabled at compile time as described in Section 1.3.

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 2. The events generated from a ‘write attributes’ request are further described in Chapter 3.

1. On Source Node

In order to send a ‘write attributes’ request, the application on the source node 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]

2. On Destination Node

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

1. For each attribute to be written, 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. If tasks within the application are not cooperative, the ZCL 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 information on mutexes, refer to Appendix A.

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: *If an ‘undivided write attributes’ request is received, an individual failed write would 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. If tasks within the application are not cooperative, the ZCL 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

On receiving an optional ‘write attributes’ response, the ZCL 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.

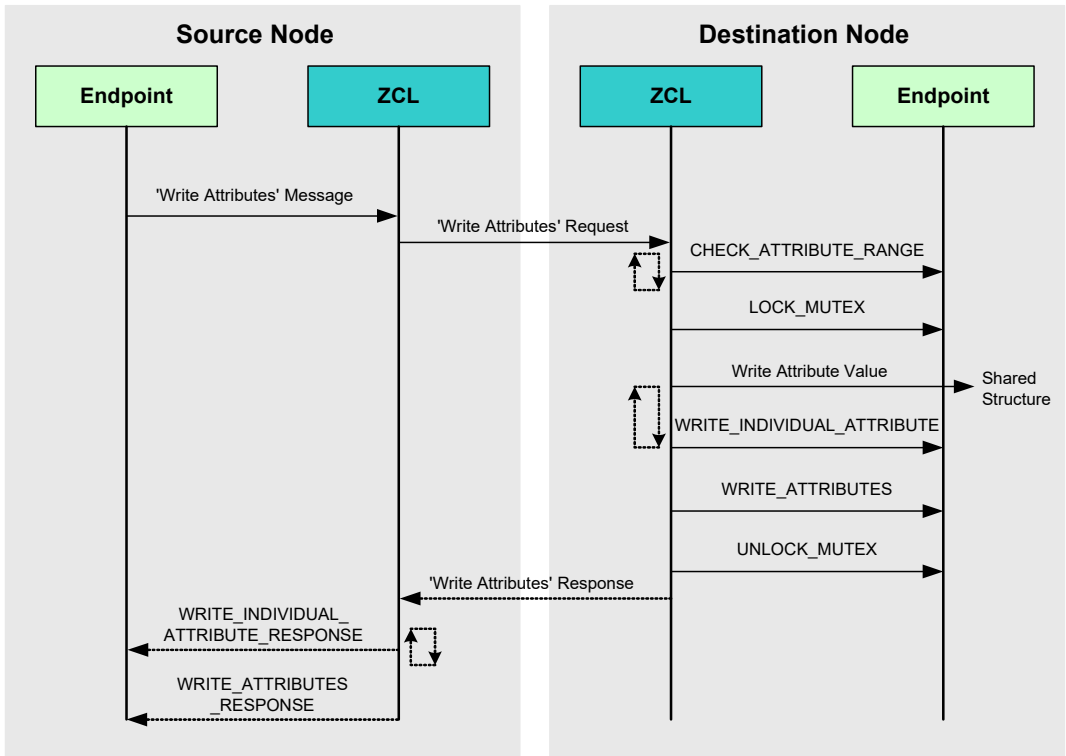


Figure 3. ‘Write Attributes’ Request and Response

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 Chapter 3.*

2.3.3.2 Writing an Attribute Value to a Local Cluster

An individual attribute of a cluster on the local node can be written to using the function **eZCL_WriteLocalAttributeValue()**. The function is blocking, returning only once the value has been written.

2.3.4 Attribute Discovery

A ZigBee cluster may have mandatory and/or optional attributes. The desired optional attributes are enabled in the cluster structure. An application running on a cluster client may need to discover which optional attributes are supported by the cluster server.

The ZCL provides functionality to perform the necessary 'attribute discovery', as described in the rest of this section.

Note:

1. *'Extended' attribute discovery is also available. When this optional attribute is used, the accessibility of each reported attribute is also indicated. This is described in Appendix C.*
2. *Alternatively, the application on a cluster client can check whether a particular attribute exists on the cluster server by attempting to read the attribute (see Section 2.3.2) - if the attribute does not exist on the server, an error is returned.*

Compile-time Options

If required, the attribute discovery feature must be explicitly enabled on the cluster server and client at compile time by including the relevant defines, from those below, in the **zcl_options.h** files:

```
#define ZCL_ATTRIBUTE_DISCOVERY_SERVER_SUPPORTED
#define ZCL_ATTRIBUTE_DISCOVERY_EXTENDED_SERVER_SUPPORTED
#define ZCL_ATTRIBUTE_DISCOVERY_CLIENT_SUPPORTED
#define ZCL_ATTRIBUTE_DISCOVERY_EXTENDED_CLIENT_SUPPORTED
```

Application Coding

The application on a cluster client can initiate a discovery of the attributes on the cluster server by calling the function **eZCL_SendDiscoverAttributesRequest()**, which sends a 'discover attributes' request to the server. This function allows a range of attributes to be searched for, defined by:

- The 'start' attribute in the range (the attribute identifier must be specified)
- The number of attributes in the range

Initially, the start attribute should be set to the first attribute of the cluster. If the discovery request does not return all the attributes used on the cluster server, the above function should be called again with the start attribute set to the next 'undiscovered' attribute. Multiple function calls may be required to discover all of the attributes used on the server.

On receiving a discover attributes request, the server handles the request automatically (provided that attribute discovery has been enabled in the compile-time options - see above) and replies with a 'discover attributes' response containing the requested information.

The arrival of this response at the client results in an E_ZCL_CBET_DISCOVER_INDIVIDUAL_ATTRIBUTE_RESPONSE event for each attribute reported in the response. Therefore, multiple events normally result from a single discover attributes request. This event contains details of the reported attribute in a `tsZCL_AttributeDiscoveryResponse` structure (see Section 6.1.10).

Following the event for the final attribute reported, the event E_ZCL_CBET_DISCOVER_ATTRIBUTES_RESPONSE is generated to indicate that all attributes from the discover attributes response have been reported.

2.3.5 Attribute Reporting

A cluster client can poll the value of an attribute on the cluster server by sending a ‘read attributes’ request, as described in Section 2.3.2. Alternatively, the server can issue unsolicited attribute reports to the client using the ‘attribute reporting’ feature (in which case there is no need for the client to request attribute values).

The attribute reporting mechanism reduces network traffic compared with the polling method. It also allows a sleeping server to report its attribute values while it is awake. Attribute reporting is an optional feature and is not supported by all devices.

An ‘attribute report’ (from server to client) can be triggered in one of the following ways:

- by the user application (on the server device)
- automatically (triggered by a change in the attribute value or periodically)

Automatic attribute reporting for an attribute can be enabled and configured remotely from the client or, for some attributes, locally on the server (see below). If it is required, automatic attribute reporting must be enabled at compile-time on both the cluster server and client. Automatic attribute reporting is more fully described in Appendix B.1 and the configuration of attribute reporting is detailed in Appendix B.3.

The ZCL specification states that certain attributes of a cluster must be reportable. Attribute reporting for these attributes remains optional but can be enabled for the individual attributes using a flag (E_ZCL_AF_RP) in the attribute definition structure - see the example code for the On/Off cluster in Section 2.3.1. This defines those attributes that the cluster server will report by default, known as ‘default reporting’, but reports on other attributes can be requested/configured by the cluster client.

Note: *Attribute reporting configuration data should be preserved in Non-Volatile Memory (NVM) to allow automatic attribute reporting to resume following a reset of the server device. Persisting this data in NVM is described in Appendix B.7.*

An attribute report can be issued directly by the server application as follows:

- For all reportable attributes using the function **eZCL_ReportAllAttributes()**
- For an individual reportable attribute using the function **eZCL_ReportAttribute()**

Only standard attributes can be reported (this does not include manufacturer-specific attributes) and only those attributes for which reporting has been enabled. This method of attribute reporting does not require any configuration, apart from enabling reports for the desired attributes. In this case, attribute reporting does not need to be enabled at compile-time on the server, but it still needs to be enabled at compile-time on the client to allow the client to receive attribute reports.

Sending an attribute report from the server is further described in Appendix B.4 and receiving an attribute report on the client is described in Appendix B.5.

2.4 Global Attributes

There are two global attributes that are used in multiple clusters. These attributes are additions to the ZCL r6 for ZigBee 3.0 and are described below.

ClusterRevision

The `ClusterRevision` global attribute is mandatory in all clusters. It indicates the revision of the cluster used by the current instance of the cluster on the local endpoint. The cluster specification from the ZCL r6 acts as the baseline for the revision numbering of a cluster - this is revision 1 of the cluster. The cluster revision number is incremented by one for each subsequent update of the cluster specification. Those cluster specifications that pre-date the ZCL r6 have assumed revision numbers of 0. To check that the local cluster instance is interoperable with a remote instance of the same cluster (perhaps based on a different cluster revision), the `ClusterRevision` attribute on the remote node should be read - if the remote cluster instance is based

on an earlier cluster revision that does not support an essential feature, the two cluster instances will not be interoperable.

AttributeReportingStatus

The `AttributeReportingStatus` global attribute is optional and used only when attribute reporting is enabled for the cluster (see Section 2.3.5). Where 'attribute report' messages are generated for multiple attributes, this attribute indicates whether there are reports pending (0x00) or the reports are complete (0x01).

2.5 Default Responses

The ZCL provides a default response which is generated in reply to a unicast command in the following circumstances:

- When there is no other relevant response and the requirement for default responses has not been disabled on the endpoint that sent the command.
- When an error results from a unicast command and there is no other relevant response, *even if the requirement for default responses has been disabled on the endpoint that sent the command*.

The default response disable setting is made in the `bDisableDefaultResponse` field of the structure `tsZCL_EndPointDefinition` detailed in Section 6.1.1. This setting dictates the value of the 'disable default response' bit in messages sent by the endpoint. The receiving device then uses this bit to determine whether to return a default response to the source device.

The default response includes the ID of the command that triggered the response and a status field (see Section 6.1.9). Therefore, in the case of an error, the command ID field of the default response contains the the identity of the command that caused the error.

Note: *The default response can be generated on reception of all commands, including responses (for example, a 'read attributes' response) but not other default responses.*

2.6 Handling Commands for Unsupported Clusters

A node might receive a cluster-specific command or general command for a cluster that is not supported. In such a case, the ZCL sends a 'default response' containing the status code E_ZCL_CMDS_UNSUPPORTED_CLUSTER to the originator of the command. This is the standard method of handling the unsupported command (as described in the ZCL specification). Default responses are described in Section 2.5.

The NXP implementation of the ZCL provides an alternative method for dealing with commands for unsupported clusters. A user-defined callback function can be introduced which is invoked when a command is received for an unsupported cluster. This function determines whether the application will handle the command and returns a Boolean value:

- If the callback function returns TRUE, the ZCL passes the unsupported command to the application in an appropriate event. For example, consider the case when a Report Attribute command is received for the Occupancy Sensing cluster which is not supported by the device. If the callback function opts to allow the application to handle this command, the callback function returns TRUE and the ZCL then passes the command to the main application in the event *E_ZCL_CBET_REPORT_INDIVIDUAL_ATTRIBUTE* or *E_ZCL_CBET_REPORT_ATTRIBUTES*, as appropriate.
- If the callback function returns FALSE, the ZCL handles the unsupported command in the standard way by sending a default response containing the status *E_ZCL_CMDS_UNSUPPORTED_CLUSTER*. The application is not notified about the received command.

The prototype for the user-defined callback function is as follows:

bool_t bZCL_OverrideHandlingEntireProfileCmd(uint16 *u16ClusterId***);**

where *u16ClusterId* is the ZigBee identifier of the cluster to which the command relates.

This callback function can be registered with the ZCL using the function **vZCL_RegisterHandleGeneralCmd CallBack()**, detailed in Section 5.1.

2.7 Handling Commands from Other Manufacturers

Every manufacturer of ZigBee Certified Products is allocated a manufacturer code by the ZigBee Alliance. The manufacturer code for NXP is 0x1037. A manufacturer-specific command that is sent by a node contains the manufacturer code for the node manufacturer.

By default, the NXP implementation of the ZCL rejects manufacturer-specific commands containing manufacturer codes other than NXP's own code. For a rejected command, the ZCL sends a 'default response' containing the status code E_ZCL_CMDS_UNSUP_MANUF_CLUSTER_COMMAND to the originator of the command. Default responses are described in Section 2.5.

However, a mechanism is available to handle multiple manufacturer codes. A user-defined callback function can be introduced, which is invoked when a manufacturer-specific command is received containing a non-NXP manufacturer code. This function determines whether the application handles the command and returns a Boolean value:

- If the callback function returns TRUE, the ZCL passes the command to the application in an appropriate event.
- If the callback function returns FALSE, the ZCL handles the command with in the standard way by sending a default response containing the status E_ZCL_CMDS_UNSUP_MANUF_CLUSTER_COMMAND. The application is not notified about the received command.

The prototype for the user-defined callback function is as follows:

bool_t bZCL_IsManufacturerCodeSupported(uint16 *u16ManufacturerCode***);**

where *u16ManufacturerCode* is the manufacturer code in the received command.

This callback function can be registered with the ZCL using the function **vZCL_RegisterCheckForManufCode CallBack()**, detailed in Section 5.1.

2.8 Bound Transmission Management

ZigBee PRO provides the facility for bound transfers/transmissions. In this case, a source endpoint on one node is bound to one or more destination endpoints on other nodes. Data sent from the source endpoint is then automatically transmitted to all the bound endpoints (without the need to specify destination addresses). The bound transmission is handled by a Bind Request Server on the source node. Binding, bound transfers, and the Bind Request Server are fully described in the *ZigBee 3.0 Stack User Guide (JNUG3130)*.

Congestion may occur if a new bound transmission is requested while the Bind Request Server is still busy completing the previous bound transmission (still sending packets to bound nodes). This causes the new bound transmission to fail. The ZCL software incorporates a feature for managing bound transmission requests, so not to overload the Bind Request Server and cause transmissions to fail.

Note: *The alternative to using this feature is for the application to re-attempt bound transmissions that fail.*

If this feature is enabled and a bound transmission request submitted to the Bind Request Server fails, the bound transmission APDU is automatically put into a queue. A one-second scheduler periodically takes the APDU at the head of the queue and submits it to the Bind Request Server for transmission. If this bound transmission also fails, the APDU is returned to the bound transmission queue.

The bound transmission queue has the following properties:

- Number of buffers in the queue

- Size of each buffer, in bytes

The feature is enabled and the above properties are defined at compile-time, as described below.

Note: *If a single APDU does not fit into a single buffer in the queue, it is stored in multiple buffers (provided that enough buffers are available).*

Compile-time Options

In order to use the bound transmission management feature, the following definitions are required in the **zcl_options.h** file.

Add this line to enable the bound transmission management feature:

```
#define CLD_BIND_SERVER
```

Add this line to define the number of buffers in the bound transmission queue (in this example, the queue will contain four buffers):

```
#define MAX_NUM_BIND_QUEUE_BUFFERS  4
```

Add this line to define the size, in bytes, of a buffer in the bound transmission queue (in this example, the buffer size is 60 bytes):

```
#define MAX_PDU_BIND_QUEUE_PAYLOAD_SIZE   60
```

Certain clusters and the ‘attribute reporting’ feature allow APS acknowledgements to be disabled for bound transmissions. The required definitions are detailed in the cluster-specific compile-time options.

2.9 Command Discovery

The ZCL provides the facility to discover the commands that a cluster instance on a remote device can receive and generate. This is useful since an individual cluster instance may not be able to receive or generate all of the commands that are theoretically supported by the cluster.

The commands that are supported by a cluster (and that can therefore potentially be discovered) are defined in a Command Definition table which is enabled in the cluster definition when Command Discovery is enabled (see Section 6.1.2).

Two ZCL functions are provided to implement the Command Discovery feature (as indicated in Section 2.9.1 below and fully described in Section 5.3).

2.9.1 Discovering Command Sets

The commands supported by a remote cluster instance can be discovered as described below.

Discovering commands that can be received

The commands that can be received by an instance of a cluster on a remote device can be discovered using the function

eZCL_SendDiscoverCommandReceivedRequest()

This function sends a request to the remote cluster instance, which responds with a list of commands (identified by their Command IDs). On receiving this response, the following events are generated on the local device:

- **E_ZCL_CBET_DISCOVER_INDIVIDUAL_COMMAND_RECEIVED_RESPONSE**

This event is generated for each individual command reported in the response. The reported information is contained in a structure of the type `tsZCL_CommandDiscoveryIndividualResponse` (see Section 6.1.17).

- **E_ZCL_CBET_DISCOVER_COMMAND_RECEIVED_RESPONSE**
  This event is generated after all the above individual events, in order to indicate the end of these events. The reported information is contained in a structure of the type `tsZCL_CommandDiscoveryResponse` (see Section 6.1.18).

Discovering commands that can be generated

The commands that can be generated by an instance of a cluster on a remote device can be discovered using the function

eZCL_SendDiscoverCommandGeneratedRequest()

This function sends a request to the remote cluster instance, which responds with a list of commands (identified by their Command IDs). On receiving this response, the following events are generated on the local device:

- **E_ZCL_CBET_DISCOVER_INDIVIDUAL_COMMAND_GENERATED_RESPONSE**
  This event is generated for each individual command reported in the response. The reported information is contained in a structure of the type `tsZCL_CommandDiscoveryIndividualResponse` (see Section 6.1.17).
- **E_ZCL_CBET_DISCOVER_COMMAND_GENERATED_RESPONSE**
  This event is generated after all the above individual events, in order to indicate the end of these events. The reported information is contained in a structure of the type `tsZCL_CommandDiscoveryResponse` (see Section 6.1.18).

Note: *The above functions can be called multiple times to discover the commands in stages. After each call, the `tsZCL_CommandDiscoveryResponse` structure contains a Boolean flag which indicates whether there are more commands to be discovered (see Section 6.1.18). For complete details, refer to the function descriptions in Section 5.3.*

2.9.2 Compile-time Options

If required, the Command Discovery feature must be enabled at compile-time.

To enable the feature, the following must be defined at both the local and remote ends:

```
#define ZCL_COMMAND_DISCOVERY_SUPPORTED
```

To enable the handling of Command Discovery requests and the generation of responses at the remote end, the following must be defined on the remote device:

```
#define ZCL_COMMAND_RECEIVED_DISCOVERY_SERVER_SUPPORTED
#define ZCL_COMMAND_GENERATED_DISCOVERY_SERVER_SUPPORTED
```

To enable the handling of Command Discovery responses at the local end, the following must be defined on the local device:

```
#define ZCL_COMMAND_RECEIVED_DISCOVERY_CLIENT_SUPPORTED
#define ZCL_COMMAND_GENERATED_DISCOVERY_CLIENT_SUPPORTED
```

3 Event Handling

This chapter describes the event handling framework which allows the ZCL to deal with stack-related and timer-related events (including cluster-specific events).

A message arriving in a message queue triggers a stack event whereas a timer event is triggered when a software timer expires (for more information on timer events, refer to Section 5.2).

The event must be wrapped in a `tsZCL_CallBackEvent` structure by the application (see Section 3.1 below), which then passes this event structure into the ZCL using the function **vZCL_EventHandler()**, described in Section 5.1. The ZCL processes the event and, if necessary, invokes the relevant endpoint callback function. Refer to Section 3.2 for more details of event processing.

3.1 Event Structure

The `tsZCL_CallBackEvent` structure, in which an event is wrapped, is as follows:

```
typedef struct
{
    teZCL_CallBackEventType         eEventType;
    uint8                           u8TransactionSequenceNumber;
    uint8                           u8EndPoint;
    teZCL_Status                    eZCL_Status;
union {
    tsZCL_IndividualAttributesResponse         sIndividualAttributeResponse;
    tsZCL_DefaultResponse                      sDefaultResponse;
    tsZCL_TimerMessage                         sTimerMessage;
    tsZCL_ClusterCustomMessage                 sClusterCustomMessage;
    tsZCL_AttributeReportingConfigurationRecord
                                sAttributeReportingConfigurationRecord;
    tsZCL_AttributeReportingConfigurationResponse
                                sAttributeReportingConfigurationResponse;
    tsZCL_AttributeDiscoveryResponse         sAttributeDiscoveryResponse;
                ZCL_AttributeStatusRecord sReportingConfigurationResponse;
    tsZCL_ReportAttributeMirror sReportAttributeMirror;
    uint32              u32TimerPeriodMs;
    tsZCL_CommandDiscoveryIndividualResponse
                        sCommandsReceivedDiscoveryIndividualResponse;
    tsZCL_CommandDiscoveryResponse      sCommandsReceivedDiscoveryResponse;
    tsZCL_CommandDiscoveryIndividualResponse
                           sCommandsGeneratedDiscoveryIndividualResponse;
    tsZCL_CommandDiscoveryResponse     sCommandsGeneratedDiscoveryResponse;
    tsZCL_AttributeDiscoveryExtendedResponse
                                       sAttributeDiscoveryExtenedResponse;
}uMessage ;
    ZPS_tsAfEvent         *pZPSevent;
    tsZCL_ClusterInstance *psClusterInstance;
} tsZCL_CallBackEvent;
```

The fields of this structure are fully described Section 6.2.

In the `tsZCL_CallBackEvent` structure, the `eEventType` field defines the type of event being posted - the various event types are described in Section 3.3 below. The union and remaining fields are each relevant to only specific event types.

3.2 Processing Events

This section outlines how the application should deal with stack events and timer events that are generated externally to the ZCL. A cluster-specific event initially arrives as one of these events.

The occurrence of an event in the ZCL queue activates a ZCL user function. The following actions must then be performed in the application:

1. The task checks whether the event that has occurred is a timer event (timer messages are collected by a user-defined function).

2. The task sets fields of the event structure `tsZCL_CallBackEvent` (see Section 3.1), as follows (all other fields are ignored):

- For a timer event, sets the field `eEventType` to E_ZCL_CBET_TIMER.
- For a millisecond timer event, sets the field `eEventType` to E_ZCL_CBET_TIMER_MS.
- For a stack event, sets the field `eEventType` to E_ZCL_ZIGBEE_EVENT and sets the field `pZPSevent` to point to the `ZPS_tsAfEvent` structure received by the application. This structure is defined in the *ZigBee 3.0 Stack User Guide (JNUG3130)*.

3. The task passes this event structure to the ZCL using **vZCL_EventHandler()** - the ZCL then identifies the event type (see Section 3.3) and invokes the appropriate endpoint callback function.

Note: *For a cluster-specific event (which arrives as a stack event or a timer event), the cluster normally contains its own event handler which is invoked by the ZCL. If the event requires the attention of the application, the ZCL replaces the `eEventType` field with E_ZCL_CBET_CLUSTER_CUSTOM and populates the `tsZCL_ClusterCustomMessage` structure with the event data. The ZCL then invokes the user-defined endpoint callback function to perform any application-specific event handling that is required.*

3.3 Events

The events that are not cluster-specific are divided into four categories (Input, Read, Write, General), as shown in the following table. The 'input events' originate externally to the ZCL and are passed into the ZCL for processing (see Section 3.2). The remaining events are generated as part of this processing.

Note: *Cluster-specific events are covered in the chapter for the relevant cluster.*

Table 13. Events

Category	Event
Input Events	E_ZCL_ZIGBEE_EVENT
	E_ZCL_CBET_TIMER
	E_ZCL_CBET_TIMER_MS
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

Table 13. Events*...continued*

Category	Event
	E_ZCL_CBET_ERROR
	E_ZCL_CBET_CLUSTER_UPDATE

The above events are described below.

Input Events

The 'input events' are generated externally to the ZCL. Such an event is received by the application, which wraps the event in a `tsZCL_CallBackEvent` structure and passes it into the ZCL using the function **vZCL_EventHandler()** - for further details of event processing, refer to Section 3.2.

- **E_ZCL_ZIGBEE_EVENT**
  All ZigBee PRO stack events to be processed by the ZCL are designated as this type of event by setting the `eEventType` field in the `tsZCL_CallBackEvent` structure to E_ZCL_ZIGBEE_EVENT.
- **E_ZCL_CBET_TIMER**
  A timer event (indicating that a timer has expired) which is to be processed by the ZCL is designated as this type of event by setting the `eEventType` field in the `tsZCL_CallBackEvent` structure to E_ZCL_CBET_TIMER.
- **E_ZCL_CBET_TIMER_MS**
  A millisecond timer event (indicating that a timer has expired) which is to be processed by the ZCL is designated as this type of event by setting the `eEventType` field in the `tsZCL_CallBackEvent` structure to E_ZCL_CBET_TIMER_MS.

Read Events

The 'read events' are generated as the result of a 'read attributes' request (see Section 2.3.2). Some of these events are generated on the remote node and some of them are generated on the local (requesting) node, as indicated in the table below.

Table 14. Read Events

Generated on local node (client):	Generated on remote node (server):
	E_ZCL_CBET_READ_REQUEST
E_ZCL_CBET_READ_INDIVIDUAL_ATTRIBUTE_RESPONSE	
E_ZCL_CBET_READ_ATTRIBUTES_RESPONSE	

The circumstances surrounding the generation of the 'read events' are outlined below:

- **E_ZCL_CBET_READ_REQUEST**
  When a 'read attributes' request has been received and passed to the ZCL (as a stack event), the ZCL generates the event E_ZCL_CBET_READ_REQUEST for the relevant endpoint to indicate that the endpoint's shared device structure is going to be read. This gives an opportunity for the application to access the shared structure first, if required - for example, to update attribute values before they are read. This event may be ignored if the application reads the hardware asynchronously - for example, driven by a timer or interrupt.
- **E_ZCL_CBET_READ_INDIVIDUAL_ATTRIBUTE_RESPONSE**
  When a 'read attributes' response has been received by the requesting node and passed to the ZCL (as a stack event), the ZCL generates the event E_ZCL_CBET_READ_INDIVIDUAL_ATTRIBUTE_RESPONSE

for each individual attribute in the response. Details of the attribute are incorporated in the structure `tsZCL_ReadIndividualAttributesResponse`, described in Section 6.2.
Note that this event is often ignored by the application, while the event E_ZCL_CBET_READ_ATTRIBUTES_RESPONSE (see next event) is handled.

- **E_ZCL_CBET_READ_ATTRIBUTES_RESPONSE**
  When a 'read attributes' response has been received by the requesting node and the ZCL has completed updating the local copy of the shared device structure, the ZCL generates the event E_ZCL_CBET_READ_ATTRIBUTES_RESPONSE. The transaction sequence number and cluster instance fields of the `tsZCL_CallBackEvent` structure are used by this event.

Write Events

The 'write events' are generated as the result of a 'write attributes' request (see Section 2.3.3). Some of these events are generated on the remote node and some of them are generated on the local (requesting) node, as indicated in the table below.

Table 15. Write Events

Generated on local node (client):	Generated on remote node (server):
	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	

During the process of receiving and processing a 'write attributes' request, the receiving application maintains a `tsZCL_IndividualAttributesResponse` structure for each individual attribute in the request:

```
typedef struct PACK {
uint16                  u16AttributeEnum;
teZCL_ZCLAttributeType  eAttributeDataType;
teZCL_CommandStatus     eAttributeStatus;
void                    *pvAttributeData;
tsZCL_AttributeStatus   *psAttributeStatus;
} tsZCL_IndividualAttributesResponse;
```

The `u16AttributeEnum` field identifies the attribute.

The field `eAttributeDataType` is set to the ZCL data type of the attribute in the request, which is checked by the ZCL to ensure that the attribute type in the request matches the expected attribute type.

The above structure is fully detailed in Section 6.2.

The circumstances surrounding the generation of the 'write events' are outlined below:

- **E_ZCL_CBET_CHECK_ATTRIBUTE_RANGE**
  When a 'write attributes' request has been received and passed to the ZCL (as a stack event), for each attribute in the request the ZCL generates the event E_ZCL_CBET_CHECK_ATTRIBUTE_RANGE for the relevant endpoint. This indicates that a 'write attributes' request has arrived and gives an opportunity for the application to do either or both of the following:
  - Check that the attribute value to be written falls within the valid range (range checking is not performed in the ZCL because the range may depend on application-specific rules).
  - Decide whether the requested write access to the attribute in the shared structure is allowed or not allowed.

- The value to be written is pointed to by `pvAttributeData` in the above structure (this does not point to the field of the shared structure containing this attribute, as the shared structure field still has its existing value).
- The attribute status field `eAttributeStatus` in the above structure is initially set to E_ZCL_SUCCESS. The application should set this field to E_ZCL_ERR_ATTRIBUTE_RANGE if the attribute value is out-of-range or to E_ZCL_DENY_ATTRIBUTE_ACCESS if it decides to disallow the write. Also note the following:
- If a conventional 'write attributes' request is received and an attribute value fails the range check or write access to an attribute is denied, this attribute is left unchanged in the shared structure but other attributes are updated.
- If an 'undivided write attributes' request is received and any attribute fails the range check or write access to any attribute is denied, no attribute values are updated in the shared structure.
- **E_ZCL_CBET_WRITE_INDIVIDUAL_ATTRIBUTE**

  Following an attempt to write an attribute value to the shared structure, the ZCL generates the event E_ZCL_CBET_WRITE_INDIVIDUAL_ATTRIBUTE for the relevant endpoint. The field `eAttributeStatus` in the structure `tsZCL_IndividualAttributesResponse` indicates to the application whether the attribute value was updated successfully:

  If the write is successful, this status field is left as E_ZCL_SUCCESS.

  If the write is unsuccessful, this status field is set to a suitable error status (see Section 7.1.4).
- **E_ZCL_CBET_WRITE_ATTRIBUTES**

  Once all the attributes in a 'write attributes' request have been processed, the ZCL generates the event E_ZCL_CBET_WRITE_ATTRIBUTES for the relevant endpoint.
- **E_ZCL_CBET_WRITE_INDIVIDUAL_ATTRIBUTE_RESPONSE**

  The E_ZCL_CBET_WRITE_INDIVIDUAL_ATTRIBUTE_RESPONSE event is generated for each attribute that is listed in an incoming 'write attributes' response message. Only attributes that have failed to be written are contained in the message. The field `eAttributeStatus` of the structure `tsZCL_IndividualAttributesResponse` indicates the reason for the failure (see Section 7.1.4).
- **E_ZCL_CBET_WRITE_ATTRIBUTES_RESPONSE**

  The E_ZCL_CBET_WRITE_ATTRIBUTES_RESPONSE event is generated when the parsing of an incoming 'write attributes' response message is complete. This event is particularly useful following a write where all the attributes have been written without errors since, in this case, no E_ZCL_CBET_WRITE_INDIVIDUAL_ATTRIBUTE_RESPONSE event is generated.

General Events

- **E_ZCL_CBET_LOCK_MUTEX and E_ZCL_CBET_UNLOCK_MUTEX**

  When an application task accesses the shared device structure of an endpoint, a mutex should be used by the task to protect the shared structure from conflicting accesses. Thus, the ZCL may need to lock or unlock a mutex in handling an event - for example, when a "read attributes" request has been received and passed to the ZCL (as a stack event). In these circumstances, the ZCL generates the following events:

  E_ZCL_CBET_LOCK_MUTEX when a mutex is to be locked

  E_ZCL_CBET_UNLOCK_MUTEX when a mutex is to be unlocked

  The ZCL specifies one of the above events in invoking the callback function for the endpoint. Thus, the endpoint callback function must include the necessary code to lock and unlock a mutex - for further information, refer to Appendix A.

  The locking and unlocking of a mutex are useful if the tasks in the application are non-cooperative while sharing the same resource. To optimize the code, the above events are not generated when the tasks are in a cooperative group. Tasks are cooperative by default and, if not required, this feature can be disabled in the **zcl_options.h** file (see Section 1.3).
- **E_ZCL_CBET_DEFAULT_RESPONSE**

  The E_ZCL_CBET_DEFAULT_RESPONSE event is generated when a ZCL default response message has been received. These messages indicate that either an error has occurred or a message

has been processed. The payload of the default response message is contained in the structure `tsZCL_DefaultResponseMessage` below:

```
typedef struct PACK {
    uint8          u8CommandId;
    uint8          u8StatusCode;
} tsZCL_DefaultResponseMessage;
```

`u8CommandId` is the ZCL command identifier of the command which triggered the default response message.

`u8StatusCode` is the status code from the default response message. It is set to 0x00 for OK or to an error code defined in the ZCL Specification.

E_ZCL_CBET_UNHANDLED_EVENT and E_ZCL_CBET_ERROR

The E_ZCL_CBET_UNHANDLED_EVENT and E_ZCL_CBET_ERROR events indicate that a stack message has been received which cannot be handled by the ZCL. The `*pZPSevent` field of the `tsZCL_CallBackEvent` structure points to the stack event that caused the event.

E_ZCL_CBET_CLUSTER_UPDATE

The E_ZCL_CBET_CLUSTER_UPDATE event indicates that one or more attribute values for a cluster on the local device may have changed.

Note: *ZCL error events and default responses (see Section 6.1.9) may be generated when problems occur in receiving commands. The possible ZCL status codes contained in the events and responses are detailed in Section 4.2.*

4 Error Handling

This chapter describes the error handling provision in the NXP implementation of the ZCL.

4.1 Last Stack Error

The last error generated by the ZigBee PRO stack can be obtained using the ZCL function **eZCL_GetLastZpsError()**, described in Section 5.1. The possible returned errors are listed in the Return/Status Codes chapter of the *ZigBee 3.0 Stack User Guide (JNUG3130)*.

4.2 Error/Command Status on Receiving Command

When a device receives a command, an error might be generated. If receiving a command results in an error, an event of the type E_ZCL_CBET_ERROR is generated on the device. In such cases, the following status codes may be used:

- The ZCL status of the event (`sZCL_CallBackEvent.eZCL_Status`) is set to one of the error codes detailed in Section 7.2.
- A 'default response' (see Section 6.1.9) may be generated which contains one of the command status codes detailed in Section 7.1.4. This response is sent to the source node of the received command (and can be intercepted using an over-air sniffer).

The table below details the error and command status codes that may be generated.

Table 16. Error and Command Status Codes

Error Status (in Event)	Command Status (in Response)	Notes
E_ZCL_ERR_ZRECEIVE_FAIL *	None	A receive error has occurred. This error is often security-based due to key establishment not being successfully completed - ZPS error is ZPS_APL_APS_E_SECURITY_FAIL.
E_ZCL_ERR_EP_UNKNOWN	E_ZCL_CMDS_SOFTWARE_FAILURE	Destination endpoint for the command is not registered with the ZCL.
E_ZCL_ERR_CLUSTER_NOT_FOUND	E_ZCL_CMDS_UNSUPPORTED_CLUSTER	Destination cluster for the command is not registered with the ZCL.
E_ZCL_ERR_SECURITY_INSUFFICIENT_FOR_CLUSTER	E_ZCL_CMDS_FAILURE	Attempt made to access a cluster using a packet without the necessary application-level (APS) encryption.
None	E_ZCL_CMDS_UNSUP_GENERAL_COMMAND	Command has no handler enabled in **zcl_options.h** file.
E_ZCL_ERR_CUSTOM_COMMAND_HANDLER_NULL_OR_RETURNED_ERR	E_ZCL_CMDS_UNSUP_CLUSTER_COMMAND	Custom command has no registered handler or its handler has not returned E_ZCL_SUCCESS.

Table 16. Error and Command Status Codes*...continued*

Error Status (in Event)	Command Status (in Response)	Notes
E_ZCL_ERR_KEY_ESTABLISH-MENT_END_POINT_NOT_FOUND	None	Key Establishment cluster has not been registered correctly.
E_ZCL_ERR_KEY_ESTABLISH-MENT_CALLBACK_ERROR	None	Key Establishment cluster callback function has returned an error.
None	E_ZCL_CMDS_MALFORMED_ COMMAND	A received message is incomplete due to some missing command-specific data.

* ZigBee PRO stack raises an error which can be retrieved using **eZCL_GetLastZpsError()**.

Part II: Common Resources

This part comprises three chapters:

- Chapter 5 details the general functions of the ZCL.
- Chapter 6 details the general structures used by the ZCL.
- Chapter 7 details the general enumerations used by the ZCL.

5 ZCL Functions

This chapter details the core functions of the ZCL that may be needed irrespective of the clusters used. These functions include:

- General functions - see Section 5.1
- Attribute Access functions - see Section 5.2
- Command Discovery functions - see Section 5.3

5.1 General Functions

This section details a set of general ZCL functions that deal with ZCL initialization, endpoint registration, timing, APS acknowledgments, event handling, and error handling:

1. eZCL_Initialise
2. eZCL_Register
3. vZCL_EventHandler
4. eZCL_Update100mS
5. vZCL_DisableAPSACK
6. eZCL_GetLastZpsError
7. vZCL_RegisterHandleGeneralCmdCallBack
8. vZCL_RegisterCheckForManufCodeCallBack

5.1.1 eZCL_Initialise

```
teZCL_Status eZCL_Initialise(
      tfpZCL_ZCLCallBackFunction cbCallBack,
      PDUM_thAPdu hAPdu);
```

Description

This function initializes the ZCL. It should be called before registering any endpoints (using one of the device-specific endpoint registration functions) and before starting the ZigBee PRO stack.

As part of this function call, you must specify a user-defined callback function that is 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 would 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

5.1.2 eZCL_Register

```
teZCL_Status eZCL_Register(
     tsZCL_EndPointDefinition *psEndPointDefinition);
```

Description

This function is used to register an endpoint with the ZCL. The function validates the clusters and corresponding attributes supported by the endpoint, and registers the endpoint.

The function should only be called to register a custom endpoint (which does not contain one of the standard ZigBee device types). It should be called for each custom endpoint on the local node. The function is not required when using a standard ZigBee device (for example, On/Off Switch) on an endpoint - in this case, the appropriate device registration function should be used.

Parameters

- *psEndPointDefinition*: Pointer to `tsZCL_EndPointDefinition` structure for the endpoint to be registered (see Section 6.1.1)

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_PARAMETER_RANGE
- E_ZCL_ERR_HEAP_FAIL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_SECURITY_RANGE
- E_ZCL_ERR_CLUSTER_0
- E_ZCL_ERR_CLUSTER_NULL
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_CLUSTER_ID_RANGE
- E_ZCL_ERR_ATTRIBUTES_NULL
- E_ZCL_ERR_ATTRIBUTE_TYPE_UNSUPPORTED,
- E_ZCL_ERR_ATTRIBUTE_NOT_FOUND,
- E_ZCL_ERR_CALLBACK_NULL

5.1.3 vZCL_EventHandler

```
void vZCL_EventHandler(
     tsZCL_CallBackEvent *psZCLCallBackEvent);
```

Description

This function should be called when an event (ZigBee stack, peripheral, timer, or cluster event) occurs. The function is used to pass the event to the ZCL. The ZCL then processes the event, including a call to any necessary callback function.

The event is passed into the function in a `tsZCL_CallBackEvent` structure, which the application must fill in - refer to Section 6.2 for details of this structure.

Parameters

- *psZCLCallBackEvent*: Pointer to a `tsZCL_CallBackEvent` event structure (see Section 6.2) containing the event to process

Returns

- None

5.1.4 eZCL_Update100mS

```
teZCL_Status eZCL_Update100mS(void);
```

Description

This function is used to service all the timing needs of the clusters used by the application and should be called every 100 ms. This can be achieved by using a 100 ms software timer to periodically prompt execution of this function.

The function calls the external user-defined function **vIdEffectTick()**, which can be used to implement an identify effect on the node. This function must be defined in the application, irrespective of whether identify effects are needed (and therefore, may be empty). The function prototype is:

```
void vIdEffectTick(void)
```

Parameters

None

Returns

E_ZCL_SUCCESS

5.1.5 vZCL_DisableAPSACK

```
void vZCL_DisableAPSACK(bool_t bDisableAPSACK);
```

Description

This function can be used to enable/disable the request of an APS acknowledgment when a ZCL command is sent. APS acknowledgments are enabled by default.

Parameters

- *bDisableAPSACK*: Enable or disable APS acknowledgment requests:
  - TRUE - Disable requests of APS acknowledgments
  - FALSE - Enable requests of APS acknowledgments

Returns

None

5.1.6 eZCL_GetLastZpsError

```
ZPS_teStatus eZCL_GetLastZpsError(void);
```

Description

This function returns the last error code generated by the ZigBee PRO stack when accessed from the ZCL.

For example, if a call to the On/Off cluster function **eCLD_OnOffCommandSend()** returns E_ZCL_ERR_ZTRANSMIT_FAIL (because the ZigBee PRO API function that was used to transmit the command failed), the **eZCL_GetLastZpsError()** function can be called to obtain the return code from the ZigBee PRO stack.

Note that the error code is not updated on a successful call to the ZigBee PRO stack. Also, there is only a single instance of the error code, so subsequent errors would over-write the current value.

Note: *If an error occurs when a command is received, an event of type E_ZCL_CBET_ERROR is generated on the receiving node. A 'default response' may also be returned to the source node of the received command. The possible ZCL status codes in the error event and in the default response are detailed in* *Section 4.2**.*

Parameters

None

Returns

The error code of the last ZigBee PRO stack error - see the *Return/Status Codes* chapter of the *ZigBee 3.0 Stack User Guide (JNUG3130)*

5.1.7 vZCL_RegisterHandleGeneralCmdCallBack

```
void vZCL_RegisterHandleGeneralCmdCallBack(void *fnPtr);
```

Description

This function is used to register an optional user-defined callback function that is invoked when a cluster-specific or general command is received for a cluster that is not supported on the local device.

The purpose of the registered callback function is to determine whether the application would handle the unsupported command. The prototype of the callback function is as follows:

bool_t bZCL_OverrideHandlingEntireProfileCmd(uint16 *u16ClusterId***);**

where *u16ClusterId* is the ZigBee identifier of the cluster to which the command relates. The function returns a Boolean value, which is TRUE if the main application handles the command and FALSE if the ZCL would handle the command:

- If the function returns TRUE, the ZCL passes the command to the main application in an appropriate event.
- If the function returns FALSE, the ZCL sends a 'default response' containing the status E_ZCL_CMDS_UNSUPPORTED_CLUSTER to the originator of the command (this is also the standard way of handling a command for an unsupported cluster when a callback function has not been registered).

For more information on handling commands for unsupported clusters, refer to Section 2.6.

Parameters

- *fnPtr*: Pointer to user-defined callback function to be registered

Returns

- None

5.1.8 vZCL_RegisterCheckForManufCodeCallBack

```
void vZCL_RegisterCheckForManufCodeCallBack(void *fnPtr);
```

Description

This function is used to register an optional user-defined callback function that is invoked when a manufacturer-specific command is received containing a manufacturer code other than NXP's own code (0x1037).

The purpose of the registered callback function is to determine whether the application would handle a received command with a given manufacturer code. The prototype of the callback function is as follows:

bool_t bZCL_IsManufacturerCodeSupported(uint16 *u16ManufacturerCode***);**

where *u16ManufacturerCode* is the manufacturer code contained in the received command. The function returns a Boolean value, which is TRUE if the main application handles the command and FALSE if the ZCL handles the command:

- If the function returns TRUE, the ZCL passes the command to the main application in an appropriate event.
- If the function returns FALSE, the ZCL sends a 'default response' containing the status E_ZCL_CMDS_UNSUP_MANUF_CLUSTER_COMMAND to the originator of the command (this is also the standard way of handling a command with a non-NXP manufacturer code when a callback function has not been registered).

For more information on handling manufacturer-specific commands containing non-NXP manufacturer codes, refer to Section 2.7.

Parameters

- *fnPtr*: Pointer to user-defined callback function to be registered

Returns

- None

5.2 Attribute Access Functions

The following functions are provided in the ZCL for accessing cluster attributes on a remote device:

1. eZCL_SendReadAttributesRequest
2. eZCL_SendWriteAttributesRequest
3. eZCL_SendWriteAttributesNoResponseRequest
4. eZCL_SendWriteAttributesUndividedRequest
5. eZCL_SendDiscoverAttributesRequest
6. eZCL_SendDiscoverAttributesExtendedRequest
7. eZCL_SendConfigureReportingCommand
8. eZCL_SendReadReportingConfigurationCommand
9. eZCL_ReportAllAttributes
10. eZCL_ReportAttribute
11. eZCL_CreateLocalReport
12. eZCL_SetReportableFlag
13. vZCL_SetDefaultReporting
14. eZCL_HandleReadAttributesResponse
15. eZCL_ReadLocalAttributeValue
16. eZCL_WriteLocalAttributeValue
17. eZCL_OverrideClusterControlFlags
18. eZCL_SetSupportedSecurity

Note: *In addition to the general function* ***eZCL_SendReadAttributesRequest()****, there are cluster-specific 'read attributes' functions for some clusters.*

5.2.1 eZCL_SendReadAttributesRequest

```
teZCL_Status eZCL_SendReadAttributesRequest(
      uint8 u8SourceEndPointId,
      uint8 u8DestinationEndPointId,
      uint16 u16ClusterId,
      bool_t bDirectionIsServerToClient,
      tsZCL_Address *psDestinationAddress,
      uint8 *pu8TransactionSequenceNumber,
      uint8 u8NumberOfAttributesInRequest,
      bool_t bIsManufacturerSpecific,
      uint16 u16ManufacturerCode,
      uint16 *pu16AttributeRequestList);
```

Description

This function can be used to send a 'read attributes' request to a cluster on a remote endpoint. Read access to cluster attributes on the remote node must be enabled at compile-time as described in Section 1.3.

Specify the endpoint on the local node from which the request is to be sent. Also specify the address of the destination node, the destination endpoint number, and the cluster from which attributes are to be read. 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: *When sending requests to multiple endpoints through a single call to this function, multiple responses would subsequently be received from the remote endpoints.*

The function allows you to read selected attributes from the remote cluster. Specify the number of attributes to be read and to identify the required attributes by means of an array of identifiers. This array must be created by the application (the memory space for the array only needs to persist for the duration of this function call). The attributes can be from the relevant ZigBee cluster specification or manufacturer-specific.

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

On receiving the 'read attributes' response, the obtained attribute values are automatically written to the local copy of the shared device structure for the remote device and an E_ZCL_CBET_READ_INDIVIDUAL_ATTRIBUTE_RESPONSE event is then generated for each attribute updated. The response may not contain values for all requested attributes. Finally, once all received attribute values have been parsed, the event E_ZCL_CBET_READ_ATTRIBUTES_RESPONSE is generated.

Parameters

- *u8SourceEndPointId*Number of the local endpoint through which the request is sent
- *u8DestinationEndPointId*Number of the remote endpoint to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP
- *u16ClusterId*Identifier of the cluster to be read (see the macros section in the cluster header file)
- *bDirectionIsServerToClient*Direction of request:
  - TRUE: Cluster server to client
  - FALSE: Cluster client to server
- *psDestinationAddress*: Pointer to a structure (see Section 6.1.4) containing the address of the remote node to which the request is sent
- *pu8TransactionSequenceNumber*: Pointer to a location to store the Transaction Sequence Number (TSN) of the request
- *u8NumberOfAttributesInRequest*: Number of attributes to be read
- *bIsManufacturerSpecific*: Indicates whether attributes are manufacturer-specific or defined in the relevant ZigBee cluster:
  - TRUE: Attributes are manufacturer-specific
  - FALSE: Attributes are from ZigBee cluster
- *u16ManufacturerCode*: ZigBee Alliance code for the manufacturer that defined proprietary attributes (set to zero if attributes are from the ZigBee cluster - that is, if *bIsManufacturerSpecific* is set to FALSE)
- *pu16AttributeRequestList*: Pointer to an array which lists the attributes to be read. The attributes are identified by using enumerations (listed in the 'Enumerations' section of each cluster-specific chapter)

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_ATTRIBUTES_0
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL
- E_ZCL_FAIL
- E_ZCL_ERR_EP_UNKNOWN

5.2.2 eZCL_SendWriteAttributesRequest

```
teZCL_Status eZCL_SendWriteAttributesRequest(
      uint8 u8SourceEndPointId,
      uint8 u8DestinationEndPointId,
      uint16 u16ClusterId,
      bool_t bDirectionIsServerToClient,
      tsZCL_Address *psDestinationAddress,
      uint8 *pu8TransactionSequenceNumber,
      uint8 u8NumberOfAttributesInRequest,
      bool_t bIsManufacturerSpecific,
               uint16 u16ManufacturerCode,
      tsZCL_WriteAttributeRecord *pu16AttributeRequestList);
```

Description

This function can be used to send a ‘write attributes’ request to a cluster on a remote endpoint. The function also demands a ‘write attributes’ response from the remote endpoint, listing any attributes that could not be updated (see below). Note that write access to cluster attributes on the remote node must be enabled at compile-time as described in Section 1.3.

You must specify the endpoint on the local node from which the request is to be sent.

You must also specify the address of the destination node, the destination endpoint number and the cluster to which attributes are to be written. 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.

The function allows you to write selected attributes to the remote cluster. You are required to specify the number of attributes to be written and to identify the required attributes by means of an array of identifiers - this array must be created by the application (the memory space for the array only needs to be valid for the duration of this function call). The attributes can be from the relevant ZigBee cluster specification or manufacturer-specific

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 is 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 a ‘write attributes’ response from the remote endpoint, the event E_ZCL_CBET_WRITE_INDIVIDUAL_ATTRIBUTE_RESPONSE is generated for each attribute that was not successfully updated on the remote endpoint. Finally, the event E_ZCL_CBET_WRITE_ATTRIBUTES_RESPONSE is generated when processing of the response is complete. If required, these events can be handled in the user-defined callback function which is specified when the (requesting) endpoint is registered using the endpoint registration function for the device type.

Parameters

- *u8SourceEndPointId*: Number of the local endpoint through which the request is sent
- *u8DestinationEndPointId*: Number of the remote endpoint to which the request is sent. This parameter is ignored while sending to address types, eZCL_AMBOUND and eZCL_AMGROUP.
- *u16ClusterId*: Identifier of the cluster to be written to (see the macros section in the cluster header file)
- *bDirectionIsServerToClient*: Direction of request:
  - TRUE: Cluster server to client
  - FALSE: Cluster client to server

- *psDestinationAddress*: Pointer to a structure (see Section 6.1.4) containing the address of the remote node to which the request is sent
- *pu8TransactionSequenceNumber*: Pointer to a location to store the Transaction Sequence Number (TSN) of the request
- *u8NumberOfAttributesInRequest*: Number of attributes to be written
- *bIsManufacturerSpecific*: Indicates whether attributes are manufacturer-specific or as defined in relevant ZigBee cluster:
  - TRUE: Attributes are manufacturer-specific
  - FALSE: Attributes are from ZigBee cluster
- *u16ManufacturerCode*: ZigBee Alliance code for the manufacturer that defined proprietary attributes (set to zero if attributes are from the ZigBee cluster - that is, if *bIsManufacturerSpecific* is set to FALSE)
- *pu16AttributeRequestList*: Pointer to an array of structures containing the attribute data to be written (see Section 6.1.21)

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_ATTRIBUTES_0
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL
- E_ZCL_FAIL
- E_ZCL_ERR_EP_UNKNOWN

5.2.3 eZCL_SendWriteAttributesNoResponseRequest

```
teZCL_Status eZCL_SendWriteAttributesNoResponseRequest(
        uint8 u8SourceEndPointId,
        uint8 u8DestinationEndPointId,
        uint16 u16ClusterId,
        bool_t bDirectionIsServerToClient,
        tsZCL_Address *psDestinationAddress,
        uint8 *pu8TransactionSequenceNumber,
        uint8 u8NumberOfAttributesInRequest,
        bool_t bIsManufacturerSpecific,
        uint16 u16ManufacturerCode,
        tsZCL_WriteAttributeRecord *pu16AttributeRequestList);
```

Description

This function can be used to send a ‘write attributes’ request to a cluster on a remote endpoint without requiring a response. If you need a response to your request, use the function **eZCL_SendWriteAttributesRequest()** instead. Note that write access to cluster attributes on the remote node must be enabled at compile-time as described in Section 1.3.

Specify the endpoint on the local node from which the request is to be sent. Also specify the address of the destination node, the destination endpoint number, and the cluster to which attributes are to be written. 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.

The function allows you to write selected attributes to the remote cluster. Users should specify the number of attributes to be written and identify the required attributes by using an array of identifiers. The application should create this array. The memory space for the array only needs to be valid for the duration of this function call. The attributes can be from the relevant ZigBee cluster specification or manufacturer-specific.

You must also provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request.

Parameters

- *u8SourceEndPointId*: Number of the local endpoint through which the request is sent.
- *u8DestinationEndPointId*: Number of the remote endpoint to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP
- *u16ClusterId*: Identifier of the cluster to be written to (see the macros section in the cluster header file):
- *bDirectionIsServerToClient*: Direction of request:
  - TRUE: Cluster server to client
  - FALSE: Cluster client to server
- *psDestinationAddress*: Pointer to a structure (see Section 6.1.4) containing the address of the remote node to which the request is sent.
- *pu8TransactionSequenceNumber*: Pointer to a location to store the Transaction Sequence Number (TSN) of the request
- *u8NumberOfAttributesInRequest*: Number of attributes to be written
- *bIsManufacturerSpecific*: : Indicates whether attributes are manufacturer-specific or as defined in relevant ZigBee cluster:
  - TRUE: Attributes are manufacturer-specific
  - FALSE: Attributes are from ZigBee cluster
- *u16ManufacturerCode*: ZigBee Alliance code for the manufacturer that defined proprietary attributes (set to zero if attributes are from the ZigBee cluster - that is, if *bIsManufacturerSpecific* is set to FALSE)
- *pu16AttributeRequestList*: Pointer to an array of structures containing the attribute data to be written (see Section 6.1.21).

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_ATTRIBUTES_0
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL
- E_ZCL_FAIL
- E_ZCL_ERR_EP_UNKNOWN

5.2.4 eZCL_SendWriteAttributesUndividedRequest

```
teZCL_Status eZCL_SendWriteAttributesUndividedRequest(
        uint8 u8SourceEndPointId,
        uint8 u8DestinationEndPointId,
        uint16 u16ClusterId,
        bool_t bDirectionIsServerToClient,
        tsZCL_Address *psDestinationAddress,
        uint8 *pu8TransactionSequenceNumber,
        uint8 u8NumberOfAttributesInRequest,
```

```
        bool_t bIsManufacturerSpecific,
         uint16 u16ManufacturerCode,
        tsZCL_WriteAttributeRecord *pu16AttributeRequestList);
```

Description

This function can be used to send an 'undivided write attributes' request to a cluster on a remote endpoint. This ensures that all the specified attributes are updated on the remote endpoint or none at all. This implies that if one of the attributes cannot be written, then none of them are updated. The function also demands a 'write attributes' response from the remote endpoint, indicating success or failure.

Note: *Write access to cluster attributes on the remote node must be enabled at compile-time as described in Section 1.3.*

You should specify the endpoint on the local node from which the request is to be sent.

You must also specify the address of the destination node, the destination endpoint number and the cluster to which attributes are to be written. 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. When sending requests to multiple endpoints through a single call to this function, multiple responses are subsequently received from the remote endpoints.

The function allows you to write selected attributes to the remote cluster. You must specify the number of attributes to be written and to identify the required attributes by means of an array of identifiers. The application should create this array, such that the memory space for the array only needs to be valid for the duration of this function call. The attributes can be from the relevant ZigBee cluster specification or manufacturer-specific

You must also provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is 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 a 'write attributes' response from the remote endpoint, the event E_ZCL_CBET_WRITE_ ATTRIBUTES_RESPONSE is generated to indicate success or failure. This event can be handled in the user-defined callback function which is specified when the (requesting) endpoint is registered using the appropriate endpoint registration function for the device type.

Parameters

- *u8SourceEndPointId*: Number of the local endpoint through which the request is sent
- *u8DestinationEndPointId*: Number of the remote endpoint to which the request is sent. Note that this parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP
- *u16ClusterId*: Identifier of the cluster to be written to (see the macros section in the cluster header file)
- *bDirectionIsServerToClient*: Direction of request:
  - TRUE: Cluster server to client
  - FALSE: Cluster client to server
- *psDestinationAddress*: Pointer to a structure (see Section 6.1.4) containing the address of the remote node to which the request is sent
- *pu8TransactionSequenceNumber*: Pointer to a location to store the Transaction Sequence Number (TSN) of the request
- *u8NumberOfAttributesInRequest*: Number of attributes to be written
- *bIsManufacturerSpecific*: Indicates whether attributes are manufacturer-specific or as defined in relevant ZigBee cluster:
  - TRUE: Attributes are manufacturer-specific
  - FALSE: Attributes are from ZigBee cluster

- *u16ManufacturerCode*: ZigBee Alliance code for the manufacturer that defined proprietary attributes (set to zero if attributes are from the ZigBee cluster - that is, if *bIsManufacturerSpecific* is set to FALSE)
- *pu16AttributeRequestList*: Pointer to an array of structures containing the attribute data to be written (see Section 6.1.21)

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_ATTRIBUTES_0
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL
- E_ZCL_FAIL
- E_ZCL_ERR_EP_UNKNOWN

5.2.5 eZCL_SendDiscoverAttributesRequest

```
teZCL_Status eZCL_SendDiscoverAttributesRequest(
        uint8 u8SourceEndPointId,
        uint8 u8DestinationEndPointId,
        uint16 u16ClusterId,
        bool_t bDirectionIsServerToClient,
        tsZCL_Address *psDestinationAddress,
        uint8 *pu8TransactionSequenceNumber,
        uint16 u16AttributeId,
        bool_t bIsManufacturerSpecific,
        uint16 u16ManufacturerCode,
        uint8 u8MaximumNumberOfIdentifiers);
```

Description

This function can be used to send a ‘discover attributes’ request to a cluster (normally a cluster server) on a remote device. The range of attributes of interest (within the standard set of cluster attributes) must be defined by specifying the identifier of the ‘start’ attribute and the number of attributes in the range. The function returns immediately and the results of the request are later received in a ‘discover attributes’ response.

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

On receiving the ‘discover attributes’ response, the event

E_ZCL_CBET_DISCOVER_INDIVIDUAL_ATTRIBUTE_RESPONSE

is generated for each attribute reported in the response. Therefore, multiple events normally result from a single function call (‘discover attributes’ request). Following the event for the final attribute reported, the event

E_ZCL_CBET_DISCOVER_ATTRIBUTES_RESPONSE

is generated to indicate that all attributes from the discover attributes response have been reported.

Attribute discovery is fully described in Section 2.3.4.

Parameters

- *u8SourceEndPointId*: Number of the local endpoint through which the request is sent
- *u8DestinationEndPointId*: Number of the remote endpoint to which the request is sent
- *u16ClusterId*: Identifier of the cluster to be queried (see the macros section in the cluster header file): :
- *bDirectionIsServerToClient*: Direction of request:
  - TRUE: Cluster server to client
  - FALSE: Cluster client to server
- *psDestinationAddress*: Pointer to a structure (see Section 6.1.4) containing the address of the remote node to which the request is sent
- *pu8TransactionSequenceNumber*: Pointer to a location to store the Transaction Sequence Number (TSN) of the request
- *u16AttributeId*: Identifier of 'start' attribute of interest
- *bIsManufacturerSpecific*: Indicates whether attributes are manufacturer-specific or as defined in relevant ZigBee cluster:
  - TRUE: Attributes are manufacturer-specific
  - FALSE: Attributes are from ZigBee cluster
- *u16ManufacturerCode*: ZigBee Alliance code for the manufacturer that defined proprietary attributes (set to zero if attributes are from the ZigBee cluster - that is, if *bIsManufacturerSpecific* is set to FALSE)
- *u8MaximumNumberOfIdentifiers*: Number of attributes in attribute range of interest (maximum number of attributes to report in response)

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_ATTRIBUTES_0
- E_ZCL_ERR_ZBUFFER_FAIL

5.2.6 eZCL_SendDiscoverAttributesExtendedRequest

```
teZCL_Status eZCL_SendDiscoverAttributesExtendedRequest(
        uint8 u8SourceEndPointId,
        uint8 u8DestinationEndPointId,
        uint16 u16ClusterId,
        bool_t bDirectionIsServerToClient,
        tsZCL_Address *psDestinationAddress,
        uint8 *pu8TransactionSequenceNumber,
        uint16 u16AttributeId,
        bool_t bIsManufacturerSpecific,
        uint16 u16ManufacturerCode,
        uint8 u8MaximumNumberOfIdentifiers);
```

Description

This function can be used to send a 'discover attributes extended' request to a cluster (normally a cluster server) on a remote device. The range of attributes of interest (within the standard set of cluster attributes) must be defined by specifying the identifier of the 'start' attribute and the number of attributes in the range. The function returns immediately and the results of the request are later received in a 'discover attributes extended' response.

Note: *An 'extended' attribute discovery is similar to a normal attribute discovery except the accessibility of each attribute is additionally indicated as being 'read', 'write' or 'reportable'.*

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is 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.

On receiving the 'discover attributes extended' response, the event

E_ZCL_CBET_DISCOVER_INDIVIDUAL_ATTRIBUTE_EXTENDED_RESPONSE

is generated for each attribute reported in the response. Therefore, multiple events normally result from a single function call ('discover attributes extended' request). Within this event, the details of the reported attribute are contained in a structure of the type `tsZCL_AttributeDiscoveryExtendedResponse` (see Section 6.1.11).

Following the event for the final attribute reported, the event

E_ZCL_CBET_DISCOVER_ATTRIBUTES_EXTENDED_RESPONSE

is generated to indicate that all attributes from the discover attributes extended response have been reported.

Extended attribute discovery is fully described in Appendix C.

Parameters

- *u8SourceEndPointId*: Number of the local endpoint through which the request is sent
- *u8DestinationEndPointId*: Number of the remote endpoint to which the request is sent
- *u16ClusterId*: Identifier of the cluster to be queried (see the macros section in the cluster header file): :
- *bDirectionIsServerToClient*: Direction of request:
  - TRUE: Cluster server to client
  - FALSE: Cluster client to server
- *psDestinationAddress*: Pointer to a structure (see Section 6.1.4) containing the address of the remote node to which the request is sent
- *pu8TransactionSequenceNumber*: Pointer to a location to store the Transaction Sequence Number (TSN) of the request
- *u16AttributeId*: Identifier of 'start' attribute of interest
- *bIsManufacturerSpecific*: Indicates whether attributes are manufacturer-specific or as defined in relevant ZigBee cluster:
  - TRUE: Attributes are manufacturer-specific
  - FALSE: Attributes are from ZigBee cluster
- *u16ManufacturerCode*: ZigBee Alliance code for the manufacturer that defined proprietary attributes (set to zero if attributes are from the ZigBee-defined cluster - that is, if *bIsManufacturerSpecific* is set to FALSE)
- *u8MaximumNumberOfIdentifiers*: Number of attributes in attribute range of interest (maximum number of attributes to report in response)

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_ATTRIBUTES_0
- E_ZCL_ERR_ZBUFFER_FAIL

5.2.7 eZCL_SendConfigureReportingCommand

```
teZCL_Status eZCL_SendConfigureReportingCommand(
        uint8 u8SourceEndPointId,
        uint8 u8DestinationEndPointId,
        uint16 u16ClusterId,
        bool_t bDirectionIsServerToClient,
        tsZCL_Address *psDestinationAddress,
        uint8 *pu8TransactionSequenceNumber,
        uint8 u8NumberOfAttributesInRequest,
        bool_t bIsManufacturerSpecific,
        uint16 u16ManufacturerCode,
        tsZCL_AttributeReportingConfigurationRecord
         *psAttributeReportingConfigurationRecord);
```

Description

This function can be used on a cluster client to send a ‘configure reporting’ command to a cluster server, in order to request automatic reporting to be configured for a set of attributes. The configuration information is provided to the function in an array of structures, where each structure contains the configuration data for a single attribute. The function will return immediately and the results of the request will later be received in a ‘configure reporting’ response.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is 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.

On receiving the ‘configure reporting’ response, the event

E_ZCL_CBET_REPORT_INDIVIDUAL_ATTRIBUTES_CONFIGURE_RESPONSE

is generated for each attribute in the response. Therefore, multiple events will normally result from a single function call (‘configure reporting’ command). Following the event for the final attribute, the event

E_ZCL_CBET_REPORT_ATTRIBUTES_CONFIGURE_RESPONSE

is generated to indicate that the configuration outcomes for all the attributes from the ‘configure reporting’ command have been reported.

Note: *In order for automatic reporting to be successfully configured for an attribute using this function, the ‘reportable flag’ for the attribute must have been set on the cluster server using the function* ***eZCL_SetReportableFlag()****.*

Attribute reporting is fully described in Appendix B.

Parameters

- *u8SourceEndPointId*: Number of the local endpoint through which the request is sent
- *u8DestinationEndPointId*: Number of the remote endpoint to which the request is sent
- *u16ClusterId*: Identifier of the cluster to be configured (see the macros section in the cluster header file)
- *bDirectionIsServerToClient*: Direction of request:
  - TRUE: Cluster server to client
  - FALSE: Cluster client to server
- *psDestinationAddress*: Pointer to a structure (see Section 6.1.4) containing the address of the remote node to which the request would be sent

- *pu8TransactionSequenceNumber*: Pointer to a location to store the Transaction Sequence Number (TSN) of the request
- *u8NumberOfAttributesInRequest*: Number of attributes for which reporting is to be configured as a result of the request
- *bIsManufacturerSpecific*: Indicates whether attributes are manufacturer-specific or as defined in relevant ZigBee cluster:
  - TRUE: Attributes are manufacturer-specific
  - FALSE: Attributes are from ZigBee cluster
- *u16ManufacturerCode*: ZigBee Alliance code for the manufacturer that defined proprietary attributes. This code is set to zero if attributes are from the ZigBee cluster - that is, if *bIsManufacturerSpecific* is set to FALSE
- *psAttributeReportingConfigurationRecord*: Pointer to array of structures, where each structure contains the attributing reporting configuration data for a single attribute (see Section 6.1.5)

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_ATTRIBUTES_0
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL
- E_ZCL_FAIL

5.2.8 eZCL_SendReadReportingConfigurationCommand

```
teZCL_Status eZCL_SendReadReportingConfigurationCommand(
        uint8 u8SourceEndPointId,
        uint8 u8DestinationEndPointId,
        uint16 u16ClusterId,
        bool_t bDirectionIsServerToClient,
        tsZCL_Address *psDestinationAddress,
        uint8 *pu8TransactionSequenceNumber,
        uint8 u8NumberOfAttributesInRequest,
        bool_t bIsManufacturerSpecific,
        uint16 u16ManufacturerCode,
        tsZCL_AttributeReadReportingConfigurationRecord
         *psAttributeReadReportingConfigurationRecord);
```

Description

This function can be used on a cluster client to send a ‘read reporting configuration’ command to a cluster server, in order to request the attribute reporting configuration data for a set of attributes. For each attribute, configuration data can be requested relating to either sending or receiving an attribute report. The required configuration data is specified to the function in an array of structures, where each structure contains the requirements for a single attribute. The function will return immediately and the results of the request will later be received in a ‘read reporting configuration’ response.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is 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.

On receiving the ‘read reporting configuration’ response, the event

E_ZCL_CBET_REPORT_READ_INDIVIDUAL_ATTRIBUTE_CONFIGURATION_RESPONSE

is generated for each attribute in the response. Therefore, multiple events will normally result from a single function call (‘read reporting configuration’ command). Following the event for the final attribute reported, the event

E_ZCL_CBET_REPORT_READ_ATTRIBUTE_CONFIGURATION_RESPONSE

is generated to indicate that the configuration outcomes for all the attributes from the ‘configure reporting’ command have been reported.

Attribute reporting is fully described in Appendix B.

Parameters

- *u8SourceEndPointId*: Number of the local endpoint through which the request is sent
- *u8DestinationEndPointId*: Number of the remote endpoint to which the request is sent.
- *u16ClusterId*: containing the attributes (see the macros section in the cluster header file)
- *bDirectionIsServerToClient*: Direction of request:
  - TRUE: Cluster server to client
  - FALSE: Cluster client to server
- *psDestinationAddress*Pointer to a structure (see Section 6.1.4) containing the address of the remote node to which the request is sent
- *pu8TransactionSequenceNumber*: Pointer to a location to store the Transaction Sequence Number (TSN) of the request
- *u8NumberOfAttributesInRequest*: Number of attributes for which reporting is to be configured as a result of the request
- *bIsManufacturerSpecific*: Indicates whether attributes are manufacturer-specific or as defined in relevant ZigBee cluster:
  - TRUE: Attributes are manufacturer-specific
  - FALSE: Attributes are from ZigBee cluster
- *u16ManufacturerCode*: ZigBee Alliance code for the manufacturer that defined proprietary attributes (set to zero if attributes are from the ZigBee cluster - that is, if *bIsManufacturerSpecific* is set to FALSE)
- *psAttributeReportingConfigurationRecord*: Pointer to an array of structures, where each structure indicates the required configuration data for a single attribute (see Section 6.1.7)

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_ATTRIBUTES_0
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL
- E_ZCL_FAIL

5.2.9 eZCL_ReportAllAttributes

```
teZCL_Status eZCL_ReportAllAttributes(
          tsZCL_Address *psDestinationAddress,
          uint16 u16ClusterID,
          uint8 u8SrcEndPoint,
          uint8 u8DestEndPoint,
```

```
            PDUM_thAPduInstance hAPduInst);
```

Description

This function can be used on the cluster server to issue an attribute report for all the reportable attributes on the server. Only the standard attributes are reported - this does not include manufacturer-specific attributes.

Use of this function requires no special configuration on the cluster server. However, the target client must be enabled to receive attribute reports (via the compile-time option ZCL_ATTRIBUTE_REPORTING_CLIENT_SUPPORTED - see Appendix B.3.1).

After this function is called and before the attribute report is sent, the event E_ZCL_CBET_REPORT_REQUEST is generated on the server, allowing the application to update the attribute values in the shared structure, if required.

Attribute reporting is fully described in Appendix B.

Parameters

- *psDestinationAddress*: Pointer to a structure (see Section 6.1.4) containing the address of the remote node to which the attribute report is sent
- *u16ClusterID*: Identifier of the cluster containing the attributes to be reported (see the macros section in the cluster header file)
- *u8SrcEndPoint*: Number of endpoint on server from which attribute report is sent
- *u8DestEndPoint*: Number of endpoint on target client to which attribute report is sent
- *hAPduInst*: Handle of APDU instance that will contain the attribute report

Returns

- E_ZCL_SUCCESS

5.2.10 eZCL_ReportAttribute

```
teZCL_Status eZCL_ReportAttribute(
        tsZCL_Address *psDestinationAddress,
        uint16 u16ClusterID,
        uint16 u16AttributeID,
        uint8 u8SrcEndPoint,
        uint8 u8DestEndPoint,
        PDUM_thAPduInstance hAPduInst);
```

Description

This function can be used on the cluster server to issue an attribute report for an individual reportable attribute on the server. Only a standard attribute can be reported - a manufacturer-specific attribute cannot be reported.

Use of this function requires no special configuration on the cluster server but the target client must be enabled to receive attribute reports (via the compile-time option ZCL_ATTRIBUTE_REPORTING_CLIENT_SUPPORTED - see Appendix B.3.1).

After this function has been called and before the attribute report is sent, the event E_ZCL_CBET_REPORT_REQUEST is generated on the server, allowing the application to update the attribute value in the shared structure, if required.

Attribute reporting is fully described in Appendix B.

Parameters

- *psDestinationAddress*: Pointer to a structure (see Section 6.1.4) containing the address of the remote node to which the attribute report is sent
- *u16ClusterID*: Identifier of the cluster containing the attribute to be reported (see the macros section in the cluster header file)
- *u16AttributeID*: Identifier of the attribute to be reported
- *u8SrcEndPoint*: Number of endpoint on server from which attribute report is sent
- *u8DestEndPoint*: Target client to which attribute report is sent
- *hAPduInst*: Handle of APDU instance that contains the attribute report

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_ATTRIBUTE_NOT_FOUND
- E_ZCL_ERR_ATTRIBUTE_NOT_REPORTABLE
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_CLUSTER_NOT_FOUND

5.2.11 eZCL_CreateLocalReport

```
teZCL_Status eZCL_CreateLocalReport(
        uint8 u8SourceEndPointId,
        uint16 u16ClusterId,
        bool_t bManufacturerSpecific,
        bool_t bIsServerAttribute,
        tsZCL_AttributeReportingConfigurationRecord
 *psAttributeReportingConfigurationRecord);
```

Description

This function can be used on a cluster server during a ‘cold start’ to register attribute reporting configuration data (with the ZCL) that has been retrieved from Non-Volatile Memory (NVM) using the Non-Volatile Memory Manager (NVM). Each call of the function registers the Attribute Reporting Configuration Record for a single attribute. This configuration record is supplied to the function in a structure that has been populated using the NVM. The function should only be called after the ZCL has been initialized. Following this function call, automatic attribute reporting can resume for the relevant attribute (for example, following a power loss or device reset).

The function must not be called for attributes that have not been configured for automatic attribute reporting. For example, it must not be used for attributes for which the maximum reporting interval is set to REPORTING_MAXIMUM_TURNED_OFF).

Attribute reporting is fully described in Appendix B.

Parameters

- *u8SourceEndPointId*: Number of endpoint on which the relevant cluster is located

- *u16ClusterId*: Identifier of the cluster containing the attribute for which retrieved attribute reporting configuration data is to be registered (see the macros section in the cluster header file)
- *bManufacturerSpecific*: Indicates whether attribute is manufacturer-specific or as defined in relevant ZigBee cluster:
  - TRUE: Attribute is manufacturer-specific
  - FALSE: Attribute is from ZigBee cluster
- *bIsServerAttribute*: Indicates whether the attribute is located on the cluster server (or client):
  - TRUE: Attribute is on cluster server
  - FALSE: Attribute is on cluster client
- *psAttributeReportingConfigurationRecord*: Pointer to structure (see Section 6.1.5) containing the reporting configuration data for the attribute

Returns

- E_ZCL_SUCCESS

5.2.12 eZCL_SetReportableFlag

```
teZCL_Status eZCL_SetReportableFlag(
        uint8 u8SrcEndPoint,
        uint16 u16ClusterID,
        bool bIsServerClusterInstance,
        bool bIsManufacturerSpecific,
        uint16 u16AttributeId);
```

Description

This function can be used on a cluster server to set (to '1') the 'reportable flag' E_ZCL_AF_RP for an attribute. Setting this flag configures the attribute to be potentially reportable, allowing automatic reporting to be configured and implemented for the attribute. It will also allow the attribute to be reported as a result of a call to **eZCL_ReportAllAttributes()**.

The cluster on which the attribute resides must be specified. The flag will be set for the specified attribute on all endpoints, but a single endpoint must be nominated which will be used to search for the attribute definition and to check that the specified cluster has been registered with the ZCL.

Attribute reporting is fully described in Appendix B.

Parameters

- *u8SourceEndPointId*: Number of endpoint to be used to search for the attribute definition and to check the cluster
- *u16ClusterId*: Identifier of the cluster containing the attribute for which the flag is to be set: (see the macros section in the cluster header file)
- *bIsServerClusterInstance*: Type of cluster instance to be set:
  - TRUE: Cluster Server
  - FALSE: Cluster Client
- *bIsManufacturerSpecific*: Indicates whether attribute is manufacturer-specific or as defined in relevant ZigBee cluster:
  - TRUE: Attribute is manufacturer-specific
  - FALSE: Attribute is from ZigBee cluster:
- *u16AttributeId*: Identifier of attribute for which the flag is to be set

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_ATTRIBUTE_NOT_FOUND
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_EP_RANGE

5.2.13 vZCL_SetDefaultReporting

```
void vZCL_SetDefaultReporting(
        tsZCL_ClusterInstance *psClusterInstance);
```

Description

This function can be used on a cluster server to enable ‘default reporting’ for those attributes that are reportable. It should be called immediately after the cluster instance has been created.

The function checks which attributes are potentially reportable - that is, which attributes have the ‘reportable flag’ E_ZCL_AF_RP set. It then sets the ‘default reporting flag’ E_ZCL_ACF_RP for these attributes.

Note: *The flag E_ZCL_AF_RP can be set for an attribute in the attribute definition or through a call to the function* ***eZCL_SetReportableFlag()****.*

Attribute reporting is fully described in Appendix B.

Parameters

- *psClusterInstance* Pointer to structure containing information about the cluster instance for which default reporting is to be enabled (see Section 6.1.16).

Returns

- None

5.2.14 eZCL_HandleReadAttributesResponse

```
teZCL_Status eZCL_HandleReadAttributesResponse(
        tsZCL_CallBackEvent *psEvent,
        uint8 *pu8TransactionSequenceNumber);
```

Description

This function can be used to examine the response to a 'read attributes' request for a remote cluster and determine whether the response is complete - that is, whether the 'read attributes' response contains all the relevant attribute values (it may be incomplete if the returned data is too large to fit into a single APDU).

eZCL_HandleReadAttributesResponse() should normally be included in the user-defined callback function that is invoked on generation of the event E_ZCL_CBET_READ_ATTRIBUTES_RESPONSE. The callback function must pass the generated event into **eZCL_HandleReadAttributesResponse()**.

If the 'read attributes' response is not complete, the function will re-send 'read attributes' requests until all relevant attribute values have been received.

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 is 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

- *psEvent* Pointer to generated event of the type E_ZCL_CBET_READ_ATTRIBUTES_RESPONSE
- *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

5.2.15 eZCL_ReadLocalAttributeValue

```
ZPS_teStatus eZCL_ReadLocalAttributeValue(
        uint8 u8SourceEndPointId,
        uint16 u16ClusterId,
        bool         bIsServerClusterInstance,
        bool bIsManufacturerSpecific,
        bool_t blsClientAttribute,
        uint16 u16AttributeId,
        void *pvAttributeValue);
```

Description

This function can be used to read a local attribute value of the specified cluster on the specified endpoint. Before reading the attribute value, the function checks that the attribute and cluster actually reside on the endpoint.

Parameters

- *u8SourceEndPointId* Number of the local endpoint on which the read will be performed
- *u16ClusterId* Identifier of the cluster to be read (see the macros section in the cluster header file)
- *bIsServerClusterInstance* Type of cluster instance to be read:
- TRUE: Cluster server
- FALSE: Cluster client
- *bIsManufacturerSpecific* Indicates whether attribute is manufacturer-specific or as defined in relevant ZigBee cluster:
- TRUE: Attribute is manufacturer-specific

- FALSE: Attribute is from ZigBee cluster
- *bIsClientAttribute* Type of attribute to be read (client or server):
- TRUE: Client attribute
- FALSE: Server attribute
- *u16AttributeId* Identifier of the attribute to be read
- *pvAttributeValue* Pointer to location to receive the read attribute value

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

5.2.16 eZCL_WriteLocalAttributeValue

```
ZPS_teStatus eZCL_WriteLocalAttributeValue(
        uint8 u8SourceEndPointId,
        uint16 u16ClusterId,
        bool         bIsServerClusterInstance,
        bool bIsManufacturerSpecific,
        bool_t blsClientAttribute,
        uint16 u16AttributeId,
        void *pvAttributeValue);
```

Description

This function writes a value to a local attribute value of the specified cluster on the specified endpoint. Before writing the attribute value, the function checks that the attribute and cluster actually reside on the endpoint.

Parameters

- *u8SourceEndPointId*: Number of the local endpoint on which the write is performed
- *u16ClusterId*: Identifier of the cluster to be written to (see the macros section in the cluster header file):
- *bIsServerClusterInstance*: Type of cluster instance to be written to:
  - TRUE: Cluster server
  - FALSE: Cluster client
- *bIsManufacturerSpecific*: Indicates whether attribute is manufacturer-specific or as defined in relevant ZigBee cluster:
  - TRUE: Attribute is manufacturer-specific
  - FALSE: Attribute is from ZigBee cluster
- *blsClientAttribute*: Type of attribute to be written to (client or server):
  - TRUE: Client attribute
  - FALSE: Server attribute

- *u16AttributeId*: Identifier of the attribute to be written to
- *pvAttributeValue*: Pointer to location containing the attribute value to be written

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

5.2.17 eZCL_OverrideClusterControlFlags

```
teZCL_Status eZCL_OverrideClusterControlFlags(
       uint8 u8SrcEndpoint,
       uint16 u16ClusterId,
       bool bIsServerClusterInstance,
       uint8 u8ClusterControlFlags);
```

Description

This function can be used to over-ride the control flag setting for the specified cluster (it can be used for any cluster). If required, this function can be called immediately after the relevant endpoint registration function (for example, for a Light Sensor device, **eHA_RegisterLightSensorEndPoint()**) or at any subsequent point in the application.

In particular, this function can be used by the application to change the default security level for a cluster.

Parameters

- *u8SourceEndPointId*: Number of the local endpoint on which the control flag is to be over-ridden
- *u16ClusterId*: Identifier of the cluster to have control flag over-ridden (see the macros section in the cluster header file):
- *bIsServerClusterInstance*: Type of cluster instance:
  - TRUE: Cluster server
  - FALSE: Cluster client
- *u8ClusterControlFlags*: Value to be written to control flag, one of:
  - E_ZCL_SECURITY_NETWORK
  - E_ZCL_SECURITY_APPLINK

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_PARAMETER_NULL

5.2.18 eZCL_SetSupportedSecurity

```
teZCL_Status eZCL_SetSupportedSecurity(
     teZCL_ZCLSendSecurity eSecuritySupported);
```

Description

This function can be used to set the security level for future transmissions from the local device. The possible levels are:

- Application-level security, which uses an application link key that is unique to the pair of nodes in communication
- Network-level security, which uses a network key that is shared by the whole network

By default, application-level security is enabled. In practice, this function can be used to disable application-level security on the local device so that the device sends all future communications with only network-level security. This is useful when transmitted packets need to be easily accessed. For example, it can be used during over-air tests performed using a packet sniffer.

Parameters

- eSecuritySupportedRequired level of security, one of:
  - E_ZCL_SECURITY_NETWORK - network-level security
  - E_ZCL_SECURITY_APPLINK - application-level security

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_RANGE

5.3 Command Discovery Functions

The following functions are provided in the ZCL for performing command discovery:

- eZCL_SendDiscoverCommandReceivedRequest
- eZCL_SendDiscoverCommandGeneratedRequest

Note: *In order to use these functions, Command Discovery must be enabled in the compile-time options. For more details, refer to the introduction to Command Discovery in Section 2.9.*

5.3.1 eZCL_SendDiscoverCommandReceivedRequest

```
teZCL_Status eZCL_SendDiscoverCommandReceivedRequest(
        uint8 u8SourceEndPointId,
        uint8 u8DestinationEndPointId,
        uint16 u16ClusterId,
        bool_t bDirectionIsServerToClient,
        tsZCL_Address *psDestinationAddress,
        uint8 *pu8TransactionSequenceNumber,
        uint8 u8CommandId,
        bool_t bIsManufacturerSpecific,
        uint16 u16ManufacturerCode,
        uint8 u8MaximumNumberOfCommands);
```

Description

This function sends a request to initiate a command discovery on a remote cluster instance to obtain a list of commands that can be received by the cluster instance.

Commands are represented by their Command IDs and the first Command ID from which the discovery is to start must be specified. The maximum number of commands to be reported must also be specified. This allows the function can be called multiple times to discover the commands in stages (see below).

The function also allows commands to be searched for that are associated with a particular manufacturer code. Alternatively, the manufacturer code can be searched for, along with the commands.

The target cluster returns a response containing the requested information. On receiving this response, the following events are generated on the local device:

- E_ZCL_CBET_DISCOVER_INDIVIDUAL_COMMAND_RECEIVED_RESPONSE: This event is generated for each individual command reported in the response. The reported information is contained in a structure of the type `tsZCL_CommandDiscoveryIndividualResponse` (see Section 6.1.17).
- E_ZCL_CBET_DISCOVER_COMMAND_RECEIVED_RESPONSE: This event is generated after all the above individual events, in order to indicate the end of these events. The reported information is contained in a structure of the type `tsZCL_CommandDiscoveryResponse` (see Section 6.1.18).

The `tsZCL_CommandDiscoveryResponse` structure in the last event contains a flag which indicates whether there are still commands to be discovered. If this is the case, the function can be called again with a new starting point (first Command ID).

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is 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.

Command discovery is described in Section 2.9.

Parameters

- *u8SourceEndPointId* Number of the local endpoint through which the request is sent
- *u8DestinationEndPointId*: Number of the remote endpoint (hosting the target cluster instance) to which the request is sent
- *u16ClusterId*: Identifier of the cluster for which a command discovery is requested
- *bDirectionIsServerToClient*: Boolean indicating the type of request in terms of source and target clusters:
- TRUE - server sending request to client
- FALSE - client sending request to server
- *psDestinationAddress*: Pointer to a structure (see Section 6.1.4) containing the address of the remote node to which the request is sent
- *pu8TransactionSequenceNumber*: Pointer to a location to store the Transaction Sequence Number (TSN) of the request
- *u8CommandId*: Command ID which is the starting point for the command discovery
- *bIsManufacturerSpecific*: Boolean indicating whether a manufacturer code is specified in the parameter *u16ManufacturerCode* below:
  - TRUE - *u16ManufacturerCode* is used
  - FALSE - *u16ManufacturerCode* is not used
- *u16ManufacturerCode*: A manufacturer-specific code (depends on the setting of *bIsManufacturerSpecific* above). 0xFFFF is a wildcard value indicating that the manufacturer code should be discovered along with the commands
- *u8MaximumNumberOfCommands*: Maximum number of commands to be discovered

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_PARAMETER_NULL

5.3.2 eZCL_SendDiscoverCommandGeneratedRequest

```
teZCL_Status eZCL_SendDiscoverCommandGeneratedRequest(
        uint8 u8SourceEndPointId,
        uint8 u8DestinationEndPointId,
        uint16 u16ClusterId,
        bool_t bDirectionIsServerToClient,
        tsZCL_Address *psDestinationAddress,
        uint8 *pu8TransactionSequenceNumber,
        uint8 u8CommandId,
        bool_t bIsManufacturerSpecific,
        uint16 u16ManufacturerCode,
        uint8 u8MaximumNumberOfCommands);
```

Description

This function sends a request to initiate a command discovery on a remote cluster instance to obtain a list of commands that can be generated by the cluster instance.

Commands are represented by their Command IDs and the first Command ID from which the discovery is to start must be specified. The maximum number of commands to be reported must also be specified. This allows the function can be called multiple times to discover the commands in several stages.

The function also allows commands to be searched for that are associated with a particular manufacturer code. Alternatively, the manufacturer code can be searched for, along with the commands.

The target cluster returns a response containing the requested information. On receiving this response, the following events are generated on the local device:

- E_ZCL_CBET_DISCOVER_INDIVIDUAL_COMMAND_GENERATED_RESPONSE: This event is generated for each individual command reported in the response. The reported information is contained in a structure of the type `tsZCL_CommandDiscoveryIndividualResponse` (see Section 6.1.17).
- E_ZCL_CBET_DISCOVER_COMMAND_GENERATED_RESPONSE: This event is generated after all the above individual events, in order to indicate the end of these events. The reported information is contained in a structure of the type `tsZCL_CommandDiscoveryResponse` (see Section 6.1.18).

The `tsZCL_CommandDiscoveryResponse` structure in the last event contains a flag which indicates whether there are still commands to be discovered. If this is the case, the function can be called again with a new starting point (first Command ID).

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is 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.

Command discovery is described in Section 2.9.

Parameters

- *u8SourceEndPointId* : Number of the local endpoint through which the request is sent

- *u8DestinationEndPointId*: Number of the remote endpoint (hosting the target cluster instance) to which the request is sent
- *u16ClusterId*: Identifier of the cluster for which a command discovery is requested
- *bDirectionIsServerToClient*: Boolean indicating the type of request in terms of source and target clusters:
  - TRUE - server sending request to client
  - FALSE - client sending request to server
- *psDestinationAddress*: Pointer to a structure (see Section 6.1.4) containing the address of the remote node to which the request is sent
- *pu8TransactionSequenceNumber*: Pointer to a location to store the Transaction Sequence Number (TSN) of the request
- *u8CommandId*: Command ID which is the starting point for the command discovery
- *bIsManufacturerSpecific*: Boolean indicating whether a manufacturer code is specified in the parameter *u16ManufacturerCode* below:
  - TRUE - *u16ManufacturerCode* is used
  - FALSE - *u16ManufacturerCode* is not used
- *u16ManufacturerCode*: A manufacturer-specific code (depends on the setting of *bIsManufacturerSpecific* above). 0xFFFF is a wildcard value indicating that the manufacturer code should be discovered along with the commands
- *u8MaximumNumberOfCommands*: Maximum number of commands to be discovered

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_ATTRIBUTES_0
- E_ZCL_ERR_ZBUFFER_FAIL

6 ZCL Structures

This chapter details the structures that are not specific to any particular ZCL cluster.

Note: *Cluster-specific structures are detailed in the chapters for the respective clusters.*

6.1 General Structures

6.1.1 tsZCL_EndPointDefinition

This structure defines the endpoint for an application:

```
struct tsZCL_EndPointDefinition
{
    uint8                       u8EndPointNumber;
    uint16                      u16ManufacturerCode;
    uint16                      u16ProfileEnum;
    bool_t                      bIsManufacturerSpecificProfile;
    uint16                      u16NumberOfClusters;
    tsZCL_ClusterInstance      *psClusterInstance;
    bool_t                      bDisableDefaultResponse;
    tfpZCL_ZCLCallBackFunction  pCallBackFunctions;
};
```

Where:

- `u8EndPointNumber` is the endpoint number between 1 and 240 (0 is reserved)
- `u16ManufacturerCode` is the manufacturer code (only valid when `bIsManufacturerSpecificProfile` is set to TRUE)
- `u16ProfileEnum` is the ZigBee application profile ID
- `bIsManufacturerSpecificProfile` indicates whether the application profile is proprietary (TRUE) or from the ZigBee Alliance (FALSE)
- `u16NumberOfClusters` is the number of clusters on the endpoint
- `psClusterInstance` is a pointer to an array of cluster instance structures
- `bDisableDefaultResponse` can be used to disable the requirement for default responses to be returned for commands sent from the endpoint (TRUE=disable, FALSE=enable)
- `pCallBackFunctions` is a pointer to the callback functions for the endpoint

6.1.2 tsZCL_ClusterDefinition

This structure defines a cluster used on a device:

```
typedef struct
{
    uint16                      u16ClusterEnum;
    bool_t                      bIsManufacturerSpecificCluster;
    uint8                       u8ClusterControlFlags;
    uint16                      u16NumberOfAttributes;
    tsZCL_AttributeDefinition  *psAttributeDefinition;
    tsZCL_SceneExtensionTable  *psSceneExtensionTable;
#ifdef ZCL_COMMAND_DISCOVERY_SUPPORTED
    uint8                       u8NumberOfCommands;
    tsZCL_CommandDefinition    *psCommandDefinition;
#endif
}  tsZCL_ClusterDefinition;
```

Where:

- `u16ClusterEnum` is the Cluster ID.
- `bIsManufacturerSpecificCluster` indicates whether the cluster is specific to a manufacturer (proprietary):
  - TRUE - proprietary cluster
  - FALSE - ZigBee cluster

u8ClusterControlFlags is a bitmap containing control bits in two parts, as follows:

Table 17. u8ClusterControlFlags bitmap

Bits	Description	Values
0 - 3	Type of security	Indicates the type of security key used via one of the following `teZCL_ZCLSend Security` enumerations (see Section 7.1.6): • E_ZCL_SECURITY_NETWORK • E_ZCL_SECURITY_APPLINK • E_ZCL_SECURITY_TEMP_APPLINK (this option is for internal use only)
4 - 7	Cluster mirror	Used internally to indicate whether the cluster is mirrored, as follows: • 0000b - Not mirrored • 1000b - Mirrored All other values are reserved

- `u16NumberOfAttributes` indicates the number of attributes in the cluster.
- `psAttributeDefinition` is a pointer to an array of attribute definition structures - see Section 6.1.3.
- `psSceneExtensionTable` is a pointer to a structure containing a Scene Extension table - see Section 6.1.20.
- The following optional pair of fields are related to the Command Discovery feature (see Section 2.9):
  - `u8NumberOfCommands` is the number of supported commands in the Command Definition table (see below).
  - `psCommandDefinition` is a pointer to a Command Definition table which contains a list of the commands supported by the cluster - each entry of the table contains the details of a supported command in a `tsZCL_CommandDefinition` structure (see Section 6.1.19).

6.1.3 tsZCL_AttributeDefinition

This structure defines an attribute used in a cluster:

```
struct tsZCL_AttributeDefinition
{
    uint16                  u16AttributeEnum;
    uint8                   u8AttributeFlags;
    teZCL_ZCLAttributeType  eAttributeDataType;
    uint16                  u16OffsetFromStructBase;
    uint16                  u16AttributeArrayLength;
};
```

Where:

- `u16AttributeEnum` is the Attribute ID.

- u8AttributeFlags is a 5-bit bitmap indicating the accessibility of the attribute (for details of the access types, refer to Section 2.3.1) - a bit is set to '1' if the corresponding access type is supported, as follows:

Table 18. u8AttributeFlags bitmap

Bit	Access Type
0	Read
1	Write
2	Reportable
3	Scene
4	Global
5-7	Reserved

- eAttributeDataType is the data type of the attribute - see Section 7.1.3.
- u16OffsetFromStructBase is the offset of the attribute's location from the start of the cluster.
- u16AttributeArrayLength is the number of consecutive attributes of the same type.

6.1.4 tsZCL_Address

This structure is used to specify the addressing mode and address for a communication with a remote node:

```
typedef struct PACK
{
  teZCL_AddressMode           eAddressMode;
  union {
        zuint16               u16GroupAddress;
        zuint16               u16DestinationAddress;
        zuint64               u64DestinationAddress;
        teAplAfBroadcastMode  eBroadcastMode;
    } uAddress;
} tsZCL_Address;
```

Where:

- eAddressMode is the addressing mode to be used (see Section 7.1.1).
- uAddress is a union containing the necessary address information (only one of the following must be set, depending on the addressing mode selected):
  - u16GroupAddress is the 16-bit group address for the target nodes.
  - u16DestinationAddress is the 16-bit network address of the target.
  - u64DestinationAddress is the 64-bit IEEE/MAC address of the target.
  - eBroadcastMode is the required broadcast mode (see Section 7.1.2).

6.1.5 tsZCL_AttributeReportingConfigurationRecord

This structure contains the configuration record for automatic reporting of an attribute.

```
typedef struct
{
    uint8                u8DirectionIsReceived;
    teZCL_ZCLAttributeType          eAttributeDataType;
    uint16                u16AttributeEnum;
    uint16                u16MinimumReportingInterval;
    uint16                u16MaximumReportingInterval;
```

```
    uint16                u16TimeoutPeriodField;
    tuZCL_AttributeReportable       uAttributeReportableChange;
} tsZCL_AttributeReportingConfigurationRecord;
```

Where:

- `u8DirectionIsReceived` indicates whether the record configures how attribute reports can be received or sent:
  - 0x00: Configures how attribute reports are sent by the server - the following fields are included in the message payload: `eAttributeDataType`, `u16MinimumReportingInterval`, `u16MaximumReportingInterval`, `uAttributeReportableChange`
  - 0x01: Configures how attribute reports are received by the client - `u16TimeoutPeriodField` is included in the message payload.
- `eAttributeDataType` indicates the data type of the attribute.
- `u16AttributeEnum` is the identifier of the attribute to which the configuration record relates.
- `u16MinimumReportingInterval` is the minimum time-interval, in seconds, between consecutive reports for the attribute - the value 0x0000 indicates no minimum (REPORTING_MINIMUM_LIMIT_NONE).
- `u16MaximumReportingInterval` is the time-interval, in seconds, between consecutive reports for periodic reporting - the following special values can also be set:
  - 0x0000 indicates that periodic reporting is to be disabled for the attribute (REPORTING_MAXIMUM_PERIODIC_TURNED_OFF).
  - 0xFFFF indicates that automatic reporting is to be completely disabled for the attribute (REPORTING_MAXIMUM_TURNED_OFF).
- `u16TimeoutPeriodField` is the timeout value, in seconds, for an attribute report - if the time elapsed since the last report exceeds this value (without receiving another report), it may be assumed that there is a problem with the attribute reporting - the value 0x0000 indicates that no timeout will be applied (REPORTS_OF_ATTRIBUTE_NOT_SUBJECT_TO_TIMEOUT).
- `uAttributeReportableChange` is the minimum change in the attribute value that causes an attribute report to be issued.

Note: *For successful attribute reporting, the timeout on the receiving client must be set to a higher value than the maximum reporting interval for the attribute on the sending server.*

6.1.6 tsZCL_AttributeReportingConfigurationResponse

This structure contains information from a ‘configure reporting’ response.

```
typedef struct
{
    teZCL_CommandStatus  eCommandStatus;
    tsZCL_AttributeReportingConfigurationRecord
                  sAttributeReportingConfigurationRecord;
}tsZCL_AttributeReportingConfigurationResponse;
```

Where:

- `eCommandStatus` is an enumeration representing the status from the response (see Section 7.1.4).
- `sAttributeReportingConfigurationRecord` is a configuration record structure (see Section 6.1.5), but only the fields `u16AttributeEnum` and `u8DirectionIsReceived` are used in the response.

6.1.7 tsZCL_AttributeReadReportingConfigurationRecord

This structure contains the details of a reporting configuration query for one attribute, to be included in a 'read reporting configuration' command:

```
typedef struct
{
    uint8       u8DirectionIsReceived;
    uint16      u16AttributeEnum;
} tsZCL_AttributeReadReportingConfigurationRecord;
```

Where:

- `u8DirectionIsReceived` specifies whether the required reporting configuration information details how the attribute reports are received or sent.
  - 0x00: Specifies that required information details how a report is sent by the server.
  - 0x01: Specifies that required information details how a report is received by the client.
- `u16AttributeEnum` is the identifier of the attribute to which the required reporting configuration information relates.

6.1.8 tsZCL_IndividualAttributesResponse

This structure is contained in a ZCL event of type E_ZCL_CBET_READ_INDIVIDUAL_ATTRIBUTE_RESPONSE (see Section):

```
typedef struct PACK {
    uint16                u16AttributeEnum;
    teZCL_ZCLAttributeType          eAttributeDataType;
    teZCL_CommandStatus           eAttributeStatus;
    void                *pvAttributeData;
} tsZCL_IndividualAttributesResponse;
```

Where:

- `u16AttributeEnum` identifies the attribute that has been read (the relevant enumerations are listed in the 'Enumerations' section of each cluster-specific chapter).
- `eAttributeDataType` is the ZCL data type of the read attribute (see Section 7.1.3).
- `eAttributeStatus` is the status of the read operation (0x00 for success or an error code - see Section 7.1.4 for enumerations).
- `pvAttributeData` is a pointer to the read attribute data which (if the read was successful) has been inserted by the ZCL into the shared device structure.

The above structure is contained in the `tsZCL_CallBackEvent` event structure, detailed in Section 6.2, when the field `eEventType` is set to E_ZCL_CBET_READ_INDIVIDUAL_ATTRIBUTE_RESPONSE.

6.1.9 tsZCL_DefaultResponse

This structure is contained in a ZCL event of type E_ZCL_CBET_DEFAULT_RESPONSE (see Section):

```
 typedef struct PACK {
     uint8  u8CommandId;
     uint8  u8StatusCode;
} tsZCL_DefaultResponse;
```

Where:

- u8CommandId is the ZCL identifier of the command that triggered the default response message
- u8StatusCode is the status code from the default response message (0x00 for OK or an error code defined in the ZCL Specification - see Section 4.2)

The above structure is contained in the tsZCL_CallBackEvent event structure, detailed in Section 6.2, when the field eEventType is set to E_ZCL_CBET_DEFAULT_RESPONSE.

6.1.10 tsZCL_AttributeDiscoveryResponse

This structure contains details of an attribute reported in a 'discover attributes' response. It is contained in a ZCL event of type E_ZCL_CBET_DISCOVER_INDIVIDUAL_ATTRIBUTE_RESPONSE.

```
typedef struct
{
    bool_t                bDiscoveryComplete;
    uint16                u16AttributeEnum;
     teZCL_ZCLAttributeType          eAttributeDataType;
} tsZCL_AttributeDiscoveryResponse;
```

where:

- bDiscoveryComplete indicates whether this is the final attribute from a 'discover attributes' to be reported:
  - TRUE - final attribute
  - FALSE - not final attribute
- u16AttributeEnum is the identifier of the attribute being reported
- eAttributeDataType indicates the data type of the attribute being reported (see Section 7.1.3)

The above structure is contained in the tsZCL_CallBackEvent event structure, detailed in Section 6.2, when the field eEventType is set to E_ZCL_CBET_DISCOVER_INDIVIDUAL_ATTRIBUTE_RESPONSE.

6.1.11 tsZCL_AttributeDiscoveryExtendedResponse

This structure contains details of an attribute reported in a 'discover attributes extended' response. It is contained in a ZCL event of type E_ZCL_CBET_DISCOVER_INDIVIDUAL_ATTRIBUTE_EXTENDED_RESPONSE.

```
typedef struct
{
    bool_t                  bDiscoveryComplete;
    uint16                  u16AttributeEnum;
    teZCL_ZCLAttributeType  eAttributeDataType;
    uint8                   u8AttributeFlags;
}tsZCL_AttributeDiscoveryExtendedResponse;
```

where:

- bDiscoveryComplete indicates whether this is the final attribute from a 'discover attributes' to be reported:
  - TRUE - final attribute
  - FALSE - not final attribute
- u16AttributeEnum is the identifier of the attribute being reported
- eAttributeDataType indicates the data type of the attribute being reported (see Section 7.1.3)
- u8AttributeFlags is a 5-bit bitmap indicating the accessibility of the reported attribute (for details of the access types, refer to Section 2.3.1) - a bit is set to '1' if the corresponding access type is supported, as follows:

Bit	Access Type
0	Read
1	Write
2	Reportable
3	Scene
4	Global
5-7	Reserved

The above structure is contained in the `tsZCL_CallBackEvent` event structure, detailed in Section 6.2, when the field `eEventType` is set to E_ZCL_CBET_DISCOVER_INDIVIDUAL_ATTRIBUTE_EXTENDED_RESPONSE.

6.1.12 tsZCL_ReportAttributeMirror

This structure contains information relating to a report attribute command:

```
typedef struct
{
    uint8                       u8DestinationEndPoint;
    uint16                      u16ClusterId;
    uint64                      u64RemoteIeeeAddress;
    teZCL_ReportAttributeStatus eStatus;
}tsZCL_ReportAttributeMirror;
```

where:

- `u8DestinationEndPoint` is the number of target endpoint for the attribute report (this is the endpoint on which the mirror for the device resides)
- `u16ClusterId` is the ID of the cluster for which information is to be mirrored
- `u64RemoteIeeeAddress` is the IEEE/MAC address of the target device for the attribute report (which contains the mirror for the device)
- `eStatus` indicates the status of the attribute report (see Section 7.1.5)

6.1.13 tsZCL_OctetString

This structure contains information on a ZCL octet (byte) string. This string is of the format:

Table 19. tsZCL_OctetString string

Octet Count, N	Data
(1 octet)	(N octets)

which contains N+1 octets, where the leading octet indicates the number of octets (N) of data in the remainder of the string (valid values are from 0x00 to 0xFE).

The `tsZCL_OctetString` structure incorporates this information as follows:

```
typedef struct
{
    uint8    u8MaxLength;
    uint8    u8Length;
    uint8   *pu8Data;
} tsZCL_OctetString;
```

Where:

- u8MaxLength is the maximum number of data octets in an octet string
- u8Length is the actual number of data octets (N) in this octet string
- pu8Data is a pointer to the first data octet of this string

Note that there is also a tsZCL_LongOctetString structure in which the octet count (N) is represented by two octets, thus allowing double the number of data octets.

6.1.14 tsZCL_CharacterString

This structure contains information on a ZCL character string. This string is of the format:

Table 20. tsZCL_CharacterString format

Character Data Length, L (1 byte)	Character Data (L bytes)

which contains L+1 bytes, where the leading byte indicates the number of bytes (L) of character data in the remainder of the string (valid values are from 0x00 to 0xFE). This value represents the number of characters in the string only if the character set used encodes each character using one byte (this is the case for ISO 646 ASCII but not in all character sets, for example, UTF8).

The tsZCL_CharacterString structure incorporates this information as follows:

```
typedef struct
{
    uint8    u8MaxLength;
    uint8    u8Length;
    uint8   *pu8Data;
} tsZCL_CharacterString;
```

where:

- u8MaxLength is the maximum number of character data bytes
- u8Length is the actual number of character data bytes (L) in this string
- pu8Data is a pointer to the first character data byte of this string

The string is not null-terminated and may therefore contain null characters mid-string.

Note that there is also a sZCL_LongCharacterString structure in which the character data length (L) is represented by two bytes, thus allowing double the number of characters.

6.1.15 tsZCL_ClusterCustomMessage

This structure contains a cluster custom message:

```
typedef struct {
     uint16                    u16ClusterId;
     void                      *pvCustomData;
} tsZCL_ClusterCustomMessage;
```

Where:

- u16ClusterId is the Cluster ID.
- pvCustomData is a pointer to the start of the data contained in the message.

6.1.16 tsZCL_ClusterInstance

This structure contains information about an instance of a cluster on a device:

```
struct tsZCL_ClusterInstance
{
    bool_t                       bIsServer;
    tsZCL_ClusterDefinition     *psClusterDefinition;
    void                        *pvEndPointSharedStructPtr;
    uint8                       *pu8AttributeControlBits;
    void                        *pvEndPointCustomStructPtr;
    tfpZCL_ZCLCustomcallCallBackFunction
                                 pCustomcallCallBackFunction;
};
```

where:

- `bIsServer` indicates whether the cluster instance is a server or client:
  - TRUE - server
  - FALSE - client
- `psClusterDefinition` is a pointer to the cluster definition structure - see Section 6.1.2
- `pvEndPointSharedStructPtr` is a pointer to the shared device structure that contains the cluster's attributes
- `pu8AttributeControlBits` is a pointer to an array of bitmaps, one for each attribute in the relevant cluster - for internal cluster definition use only, array should be initialized to 0
- `pvEndPointCustomStructPtr` is a pointer to any custom data (only relevant to a user-defined cluster)
- `pCustomcallCallBackFunction` is a pointer to a custom callback function (only relevant to a user-defined cluster)

6.1.17 tsZCL_CommandDiscoveryIndividualResponse

This structure contains information about an individual command reported in a Command Discovery response (see Section 2.9).

```
typedef struct
{
    uint8  u8CommandEnum;
    uint8  u8CommandIndex;
} tsZCL_CommandDiscoveryIndividualResponse;
```

where:

- `u8CommandEnum` is the Command ID of the reported command
- `u8CommandIndex` is the index of the reported command in the response payload

The above structure is contained in the `tsZCL_CallBackEvent` event structure, detailed in Section 6.2, when the field `eEventType` is set to E_ZCL_CBET_DISCOVER_INDIVIDUAL_COMMAND_RECEIVED_RESPONSE or E_ZCL_CBET_DISCOVER_INDIVIDUAL_COMMAND_GENERATED_RESPONSE.

6.1.18 tsZCL_CommandDiscoveryResponse

This structure contains information about a Command Discovery response (see Section 2.9).

```
typedef struct
{
    bool_t bDiscoveryComplete;
```

```
    uint8  u8NumberOfCommands;
} tsZCL_CommandDiscoveryResponse;
```

Where:

- `bDiscoveryComplete` is a Boolean flag which indicates whether the Command Discovery is complete, i.e. whether there are any commands remaining to be discovered:
  - TRUE - all commands have been discovered
  - FALSE - there are further commands to be discovered
- `u8NumberOfCommands` is the number of discovered commands reported in the response (the individual commands are reported in a structure of the type `tsZCL_CommandDiscoveryIndividualResponse` - see Section 6.1.17)

The above structure is contained in the `tsZCL_CallBackEvent` event structure, detailed in Section 6.2, when the field `eEventType` is set to E_ZCL_CBET_DISCOVER_COMMAND_RECEIVED_RESPONSE or E_ZCL_CBET_DISCOVER_COMMAND_GENERATED_RESPONSE.

6.1.19 tsZCL_CommandDefinition

This structure contains the details of a command which is supported by the cluster (and can be reported in Command Discovery).

```
struct tsZCL_CommandDefinition
{
     uint8   u8CommandEnum;
     uint8   u8CommandFlags;
};
```

Where:

- `u8CommandEnum` is the Command ID within the cluster
- `u8CommandFlags` is a bitmap containing a set of control flags, as follows:

Bits	Enumeration	Description
0	E_ZCL_CF_RX	Command is generated by the client and received by the server
1	E_ZCL_CF_TX	Command is generated by the server and received by the client
2	-	Reserved
3	E_ZCL_CF_MS	Command is manufacturer-specific
4 - 7	-	Reserved

6.1.20 tsZCL_SceneExtensionTable

This structure contains a Scenes Extension table.

```
typedef struct
{
    tfpZCL_SceneEventHandler    pSceneEventHandler;
    uint16                      u16NumberOfAttributes;
    uint16                      au16Attributes[];
} tsZCL_SceneExtensionTable;
```

Where:

- `pSceneEventHandler` is a pointer a Scenes event handler function

- `u16NumberOfAttributes` is the number of attributes in the Scene extension
- `au16Attributes` is an array of the attribute IDs of the attributes in the Scene extension

6.1.21 tsZCL_WriteAttributeRecord

The is structure contains the details for a ‘write attribute’ operation.

```
typedef struct
{
          teZCL_ZCLAttributeType   eAttributeDataType;
          uint16                   u16AttributeEnum;
          uint8                                                    *pu8Att
}tsZCL_WriteAttributeRecord;
```

Where:

- `eAttributeDataType` is an enumeration indicating the attribute data type (for the enumerations, refer to Section 7.1.3).
- `u16AttributeEnum` is an enumeration for the attribute identifier (for the relevant ‘Attribute ID’ enumerations, refer to the ‘Enumerations’ section of each cluster-specific chapter).
- `pu8AttributeData` is a pointer to the attribute data to be written.

6.2 Event Structure (tsZCL_CallBackEvent)

A ZCL event must be wrapped in the following `tsZCL_CallBackEvent` structure before being passed into the function **vZCL_EventHandler()**:

```
typedef struct
 {
teZCL_CallBackEventType                    eEventType;
uint8                                      u8TransactionSequenceNumber;
uint8                                      u8EndPoint;
teZCL_Status                               eZCL_Status;
union {
tsZCL_IndividualAttributesResponse         sIndividualAttributeResponse;
tsZCL_DefaultResponse                      sDefaultResponse;
tsZCL_TimerMessage                         sTimerMessage;
tsZCL_ClusterCustomMessage                 sClusterCustomMessage;
tsZCL_AttributeReportingConfigurationRecord
                                           sAttributeReportingConfigurationRecord;
tsZCL_AttributeReportingConfigurationResponse

 sAttributeReportingConfigurationResponse;
tsZCL_AttributeDiscoveryResponse           sAttributeDiscoveryResponse;
tsZCL_AttributeStatusRecord                sReportingConfigurationResponse;
tsZCL_ReportAttributeMirror                sReportAttributeMirror;
uint32                                     u32TimerPeriodMs;
tsZCL_CommandDiscoveryIndividualResponse

 sCommandsReceivedDiscoveryIndividualResponse;
tsZCL_CommandDiscoveryResponse             sCommandsReceivedDiscoveryResponse;
tsZCL_CommandDiscoveryIndividualResponse

 sCommandsGeneratedDiscoveryIndividualResponse;
tsZCL_CommandDiscoveryResponse             sCommandsGeneratedDiscoveryResponse;
tsZCL_AttributeDiscoveryExtendedResponse
                                           sAttributeDiscoveryExtenedResponse;
```

```
 }uMessage ;
      ZPS_tsAfEvent *pZPSevent;
      tsZCL_ClusterInstance *psClusterInstance;
} tsZCL_CallBackEvent;
```

where

- `eEventType`: specifies the type of event generated - see Section 7.3.
- `u8TransactionSequenceNumber` is the Transaction Sequence Number (TSN) of the incoming ZCL message (if any) which triggered the ZCL event.
- `u8EndPoint` is the endpoint on which the ZCL message (if any) was received.
- `eZCL_Status` is the status of the operation that the event reports - see Section 7.2
- `uMessage` is a union containing information that is only valid for specific events:
  - `sIndividualAttributeResponse` contains the response to a ‘read attributes’ or ‘write attributes’ request - see Section 6.1.8.
  - `sDefaultResponse` contains the response to a request (other than a read request) - see Section 6.1.9.
  - `sTimerMessage` contains the details of a timer event - this feature is included for future use.
  - `sClusterCustomMessage` contains details of a cluster custom command - see Section 6.1.15
  - `sAttributeReportingConfigurationRecord` contains the attribute reporting configuration data from the ‘configure reporting’ request for an attribute - see Section 6.1.5.
  - `sAttributeReportingConfigurationResponse` is reserved for future use.
  - `sAttributeDiscoveryResponse` contains the details of an attribute reported in a ‘discover attributes’ response - see Section 6.1.10.
  - `sReportingConfigurationResponse` is reserved for future use.
  - `sReportAttributeMirror` contains information on the device from which a ZCL ‘report attribute’ command has been received.
  - `u32TimerPeriodMs` contains the timed period of the millisecond timer which is enabled by the application when the event E_ZCL_CBET_ENABLE_MS_TIMER occurs.
  - `sCommandsReceivedDiscoveryIndividualResponse` contains information about an individual command (that can be received) reported in a Command Discovery response - see Section 6.1.17.
  - `sCommandsReceivedDiscoveryResponse` contains information about a Command Discovery response which reports commands that can be received - see Section 6.1.18.
  - `sCommandsGeneratedDiscoveryIndividualResponse` contains information about an individual command (that can be generated) reported in a Command Discovery response - see Section 6.1.17.
  - `sCommandsGeneratedDiscoveryResponse` contains information about a Command Discovery response which reports commands that can be generated - see Section 6.1.18.
  - `sAttributeDiscoveryExtenedResponse` contains information from a Discover Attributes Extended response - see Section 6.1.11.

The remaining fields are common to more than one event type but are not valid for all events:

- `pZPSevent` is a pointer to the stack event (if any) that caused the ZCL event.
- `psClusterInstance` is a pointer to the cluster instance structure that holds the information relating to the cluster being accessed.

7 Enumerations and Status Codes

This chapter details the enumerations and status codes provided in the NXP implementation of the ZCL or provided in the ZigBee PRO APIs and used by the ZCL.

7.1 General Enumerations

7.1.1 Addressing Modes (teZCL_AddressMode)

The following enumerations are used to specify the addressing mode to be used in a communication with a remote node:

```
typedef enum
{
    E_ZCL_AM_BOUND,
    E_ZCL_AM_GROUP,
    E_ZCL_AM_SHORT,
    E_ZCL_AM_IEEE,
    E_ZCL_AM_BROADCAST,
    E_ZCL_AM_NO_TRANSMIT,
    E_ZCL_AM_BOUND_NO_ACK,
    E_ZCL_AM_SHORT_NO_ACK,
    E_ZCL_AM_IEEE_NO_ACK,
    E_ZCL_AM_BOUND_NON_BLOCKING,
    E_ZCL_AM_BOUND_NON_BLOCKING_NO_ACK,
    E_ZCL_AM_ENUM_END, /* enum End */
} teZCL_AddressMode;
```

The above enumerations are described in the table below.

Table 21. Addressing Mode Enumerations

Enumeration	Description
E_ZCL_AM_BOUND	Use one or more bound nodes/endpoints, with acknowledgments
E_ZCL_AM_GROUP	Use a pre-defined group address, with acknowledgments
E_ZCL_AM_SHORT	Use a 16-bit network address, with acknowledgments
E_ZCL_AM_IEEE	Use a 64-bit IEEE/MAC address, with acknowledgments
E_ZCL_AM_BROADCAST	Perform a broadcast (see Section 7.1.2)
E_ZCL_AM_NO_TRANSMIT	Do not transmit
E_ZCL_AM_BOUND_NO_ACK	Perform a bound transmission, with no acknowledgments
E_ZCL_AM_SHORT_NO_ACK	Perform a transmission using a 16-bit network address, with no acknowledgments
E_ZCL_AM_IEEE_NO_ACK	Perform a transmission using a 64-bit IEEE/MAC address, with no acknowledgments
E_ZCL_AM_BOUND_NON_BLOCKING	Perform a non-blocking bound transmission, with acknowledgments
E_ZCL_AM_BOUND_NON_BLOCKING_NO_ACK	Perform a non-blocking bound transmission, with no acknowledgments

The required addressing mode is specified in the structure `tsZCL_Address` (see Section 6.1.4).

7.1.2 Broadcast Modes (ZPS_teAplAfBroadcastMode)

The following enumerations are used to specify the type of broadcast (when the addressing mode for a communication has been set to E_ZCL_AM_BROADCAST (see Section 7.1.1)):

```
typedef enum
{
    ZPS_E_APL_AF_BROADCAST_ALL,
    ZPS_E_APL_AF_BROADCAST_RX_ON,
    ZPS_E_APL_AF_BROADCAST_ZC_ZR
} ZPS_teAplAfBroadcastMode;
```

The above enumerations are described in the table below.

Table 22. Broadcast Mode Enumerations

Enumeration	Description
ZPS_E_APL_AF_BROADCAST_ALL	All End Devices
ZPS_E_APL_AF_BROADCAST_RX_ON	Nodes on which the radio receiver remains enabled when the node is idle (e.g. sleeping)
ZPS_E_APL_AF_BROADCAST_ZC_ZR	Only the Coordinator and Routers

The required broadcast mode is specified in the structure `tsZCL_Address` (see Section 6.1.4).

7.1.3 Attribute Types (teZCL_ZCLAttributeType)

The following enumerations are used to represent the attribute types in the/ZCL clusters:

```
typedef enum
{
    /* Null */
    E_ZCL_NULL            = 0x00,
    /* General Data */
    E_ZCL_GINT8           = 0x08,             // General 8 bit - not specified if signed
    E_ZCL_GINT16,
    E_ZCL_GINT24,
    E_ZCL_GINT32,
    E_ZCL_GINT40,
    E_ZCL_GINT48,
    E_ZCL_GINT56,
    E_ZCL_GINT64,
    /* Logical */
    E_ZCL_BOOL            = 0x10,
    /* Bitmap */
    E_ZCL_BMAP8           = 0x18,             // 8 bit bitmap
    E_ZCL_BMAP16,
    E_ZCL_BMAP24,
    E_ZCL_BMAP32,
    E_ZCL_BMAP40,
    E_ZCL_BMAP48,
    E_ZCL_BMAP56,
    E_ZCL_BMAP64,
    /* Unsigned Integer */
    E_ZCL_UINT8           = 0x20,             // Unsigned 8 bit
    E_ZCL_UINT16,
    E_ZCL_UINT24,
    E_ZCL_UINT32,
    E_ZCL_UINT40,
```

```
    E_ZCL_UINT48,
    E_ZCL_UINT56,
    E_ZCL_UINT64,
    /* Signed Integer */
    E_ZCL_INT8            = 0x28,           // Signed 8 bit
    E_ZCL_INT16,
    E_ZCL_INT24,
    E_ZCL_INT32,
    E_ZCL_INT40,
    E_ZCL_INT48,
    E_ZCL_INT56,
    E_ZCL_INT64,
    /* Enumeration */
    E_ZCL_ENUM8           = 0x30,           // 8 Bit enumeration
    E_ZCL_ENUM16,
    /* Floating Point */
    E_ZCL_FLOAT_SEMI      = 0x38,       // Semi precision
    E_ZCL_FLOAT_SINGLE,               // Single precision
    E_ZCL_FLOAT_DOUBLE,                // Double precision
    /* String */
    E_ZCL_OSTRING         = 0x41,        // Octet string
    E_ZCL_CSTRING,                  // Character string
    E_ZCL_LOSTRING,                    // Long octet string
    E_ZCL_LCSTRING,                  // Long character string
    /* Ordered Sequence */
    E_ZCL_ARRAY           = 0x48,
    E_ZCL_STRUCT          = 0x4c,
    E_ZCL_SET             = 0x50,
    E_ZCL_BAG             = 0x51,
    /* Time */
    E_ZCL_TOD             = 0xe0,            // Time of day
    E_ZCL_DATE,                              // Date
    E_ZCL_UTCT,                              // UTC Time
    /* Identifier */
    E_ZCL_CLUSTER_ID      = 0xe8,          // Cluster ID
    E_ZCL_ATTRIBUTE_ID,                  // Attribute ID
    E_ZCL_BACNET_OID,                    // BACnet OID
    /* Miscellaneous */
    E_ZCL_IEEE_ADDR       = 0xf0,        // 64 Bit IEEE Address
    E_ZCL_KEY_128,                    // 128 Bit security key
    /* Unknown */
    E_ZCL_UNKNOWN         = 0xff
} teZCL_ZCLAttributeType;
```

7.1.4 Command Status (teZCL_CommandStatus)

The following enumerations are used to indicate the status of a command:

```
typedef enum
{
    E_ZCL_CMDS_SUCCESS =0x00,
    E_ZCL_CMDS_FAILURE,
    E_ZCL_CMDS_NOT_AUTHORIZED =0x7e,
    E_ZCL_CMDS_RESERVED_FIELD_NOT_ZERO,
    E_ZCL_CMDS_MALFORMED_COMMAND =0x80,
    E_ZCL_CMDS_UNSUP_CLUSTER_COMMAND,
    E_ZCL_CMDS_UNSUP_GENERAL_COMMAND,
    E_ZCL_CMDS_UNSUP_MANUF_CLUSTER_COMMAND,
    E_ZCL_CMDS_UNSUP_MANUF_GENERAL_COMMAND,
    E_ZCL_CMDS_INVALID_FIELD,
    E_ZCL_CMDS_UNSUPPORTED_ATTRIBUTE,
    E_ZCL_CMDS_INVALID_VALUE,
    E_ZCL_CMDS_READ_ONLY,
```

```
    E_ZCL_CMDS_INSUFFICIENT_SPACE,
    E_ZCL_CMDS_DUPLICATE_EXISTS,
    E_ZCL_CMDS_NOT_FOUND,
    E_ZCL_CMDS_UNREPORTABLE_ATTRIBUTE,
    E_ZCL_CMDS_INVALID_DATA_TYPE,
    E_ZCL_CMDS_INVALID_SELECTOR,
    E_ZCL_CMDS_WRITE_ONLY,
    E_ZCL_CMDS_INCONSISTENT_STARTUP_STATE,
    E_ZCL_CMDS_DEFINED_OUT_OF_BAND,
    E_ZCL_CMDS_INCONSISTENT,
    E_ZCL_CMDS_ACTION_DENIED,
    E_ZCL_CMDS_TIMEOUT,
    E_ZCL_CMDS_HARDWARE_FAILURE =0xc0,
    E_ZCL_CMDS_SOFTWARE_FAILURE,
    E_ZCL_CMDS_CALIBRATION_ERROR,
    E_ZCL_CMDS_UNSUPPORTED_CLUSTER,
    E_ZCL_CMDS_ENUM_END
} teZCL_CommandStatus;
```

The above enumerations are described in the table below.

Table 23. Command Status Enumerations

Enumeration	Description
E_ZCL_CMDS_SUCCESS	Command was successful
E_ZCL_CMDS_FAILURE	Command was unsuccessful
E_ZCL_CMDS_NOT_AUTHORIZED	Sender does not have authorisation to issue the command
E_ZCL_CMDS_RESERVED_FIELD_NOT_ZERO	A reserved field of command is not set to zero
E_ZCL_CMDS_MALFORMED_COMMAND	Command has missing fields or invalid field values
E_ZCL_CMDS_UNSUP_CLUSTER_COMMAND	The specified cluster has not been registered with the ZCL on the device
E_ZCL_CMDS_UNSUP_GENERAL_COMMAND	Command does not have a handler enabled in the **zcl_options.h** file
E_ZCL_CMDS_UNSUP_MANUF_CLUSTER_COMMAND	Manufacturer-specific cluster command is not sup-ported or has unknown manufacturer code
E_ZCL_CMDS_UNSUP_MANUF_GENERAL_COMMAND	Manufacturer-specific ZCL command is not sup-ported or has unknown manufacturer code
E_ZCL_CMDS_INVALID_FIELD	Command has field which contains invalid value
E_ZCL_CMDS_UNSUPPORTED_ATTRIBUTE	Specified attribute is not supported on the device
E_ZCL_CMDS_INVALID_VALUE	Specified attribute value is out of range or a reserved value
E_ZCL_CMDS_READ_ONLY	Attempt to write to read-only attribute
E_ZCL_CMDS_INSUFFICIENT_SPACE	Not enough memory space to perform requested operation
E_ZCL_CMDS_DUPLICATE_EXISTS	Attempt made to create a table entry that already exists in the target table
E_ZCL_CMDS_NOT_FOUND	Requested information cannot be found
E_ZCL_CMDS_UNREPORTABLE_ATTRIBUTE	Periodic reports cannot be produced for this attribute

Table 23. Command Status Enumerations*...continued*

Enumeration	Description
E_ZCL_CMDS_INVALID_DATA_TYPE	Invalid data type specified for attribute
E_ZCL_CMDS_INVALID_SELECTOR	Incorrect selector for this attribute
E_ZCL_CMDS_WRITE_ONLY	Issuer of command does not have authorisation to read specified attribute
E_ZCL_CMDS_INCONSISTENT_STARTUP_STATE	Setting the specified values would put device into an inconsistent state on start-up
E_ZCL_CMDS_DEFINED_OUT_OF_BAND	Attempt has been made to write to attribute using an out-of-band method or not over-air
E_ZCL_CMDS_HARDWARE_FAILURE	Command was unsuccessful due to hardware failure
E_ZCL_CMDS_SOFTWARE_FAILURE	Command was unsuccessful due to software failure
E_ZCL_CMDS_CALIBRATION_ERROR	Error occurred during calibration
E_ZCL_CMDS_UNSUPPORTED_CLUSTER	The cluster is not supported

7.1.5 Report Attribute Status (teZCL_ReportAttributeStatus)

The following enumerations are used to indicate the status of a report attribute command.

```
typedef enum
{
    E_ZCL_ATTR_REPORT_OK = 0x00,
    E_ZCL_ATTR_REPORT_EP_MISMATCH,
    E_ZCL_ATTR_REPORT_ADDR_MISMATCH,
    E_ZCL_ATTR_REPORT_ERR
} teZCL_ReportAttributeStatus;
```

The above enumerations are described in the table below.

Table 24. Report Attribute Status Enumerations

Enumeration	Description
E_ZCL_ATTR_REPORT_OK	Indicates that report is valid
E_ZCL_ATTR_REPORT_EP_MISMATCH	Indicates that source endpoint does not match endpoint in mirror
E_ZCL_ATTR_REPORT_ADDR_MISMATCH	Indicates that source address does not match address in mirror
E_ZCL_ATTR_REPORT_ERR	Indicates that there is an error in the report

7.1.6 Security Level (teZCL_ZCLSendSecurity)

The following enumerations are used to indicate the security level for transmissions:

```
typedef enum
{
    E_ZCL_SECURITY_NETWORK = 0x00,
    E_ZCL_SECURITY_APPLINK,
    E_ZCL_SECURITY_TEMP_APPLINK,
    E_ZCL_SECURITY_ENUM_END
```

```
} teZCL_ZCLSendSecurity;
```

The above enumerations are described in the table below.

Table 25. Security Level Enumerations

Enumeration	Description
E_ZCL_SECURITY_NETWORK	Network-level security, using a network key
E_ZCL_SECURITY_APPLINK	Application-level security, using an application link key
E_ZCL_SECURITY_TEMP_APPLINK	Temporary application-level security. This option is for internal use only. This is used for situations in which an application link key is to be used temporarily. For example, it can be used for an individual communication.

7.2 General Return codes (ZCL Status)

The following ZCL status enumerations are returned by many API functions to indicate the outcome of the function call.

```
typedef enum
{
   // General
   E_ZCL_SUCCESS = 0x0,
   E_ZCL_FAIL,                   // 01
   E_ZCL_ERR_PARAMETER_NULL,     // 02
   E_ZCL_ERR_PARAMETER_RANGE,    // 03
   E_ZCL_ERR_HEAP_FAIL,          // 04
   // Specific ZCL status codes
   E_ZCL_ERR_EP_RANGE,           // 05
   E_ZCL_ERR_EP_UNKNOWN,         // 06
   E_ZCL_ERR_SECURITY_RANGE,     // 07
   E_ZCL_ERR_CLUSTER_0,          // 08
   E_ZCL_ERR_CLUSTER_NULL,       // 09
   E_ZCL_ERR_CLUSTER_NOT_FOUND, // 10
   E_ZCL_ERR_CLUSTER_ID_RANGE,  // 11
   E_ZCL_ERR_ATTRIBUTES_NULL,    // 12
   E_ZCL_ERR_ATTRIBUTES_0,       // 13
   E_ZCL_ERR_ATTRIBUTE_WO,       // 14
   E_ZCL_ERR_ATTRIBUTE_RO,       // 15
   E_ZCL_ERR_ATTRIBUTES_ACCESS, // 16
   E_ZCL_ERR_ATTRIBUTE_TYPE_UNSUPPORTED, // 17
   E_ZCL_ERR_ATTRIBUTE_NOT_FOUND,    // 18
   E_ZCL_ERR_CALLBACK_NULL,          // 19
   E_ZCL_ERR_ZBUFFER_FAIL,           // 20
   E_ZCL_ERR_ZTRANSMIT_FAIL,         // 21
   E_ZCL_ERR_CLIENT_SERVER_STATUS,   // 22
   E_ZCL_ERR_TIMER_RESOURCE,         // 23
   E_ZCL_ERR_ATTRIBUTE_IS_CLIENT,    // 24
   E_ZCL_ERR_ATTRIBUTE_IS_SERVER,    // 25
   E_ZCL_ERR_ATTRIBUTE_RANGE,        // 26
   E_ZCL_ERR_ATTRIBUTE_MISMATCH,     // 27
   E_ZCL_ERR_KEY_ESTABLISHMENT_MORE_THAN_ONE_CLUSTER, //28
   E_ZCL_ERR_INSUFFICIENT_SPACE,     // 29
   E_ZCL_ERR_NO_REPORTABLE_CHANGE,   // 30
   E_ZCL_ERR_NO_REPORT_ENTRIES,      // 31
   E_ZCL_ERR_ATTRIBUTE_NOT_REPORTABLE, //32
   E_ZCL_ERR_ATTRIBUTE_ID_ORDER,     // 33
   E_ZCL_ERR_MALFORMED_MESSAGE,      // 34
   E_ZCL_ERR_MANUFACTURER_SPECIFIC, // 35
```

```
    E_ZCL_ERR_PROFILE_ID,             // 36
    E_ZCL_ERR_INVALID_VALUE,          // 37
    E_ZCL_ERR_CERT_NOT_FOUND,         // 38
    E_ZCL_ERR_CUSTOM_DATA_NULL,       // 39
    E_ZCL_ERR_TIME_NOT_SYNCHRONISED, // 40
    E_ZCL_ERR_SIGNATURE_VERIFY_FAILED, //41
    E_ZCL_ERR_ZRECEIVE_FAIL,           // 42
    E_ZCL_ERR_KEY_ESTABLISHMENT_END_POINT_NOT_FOUND, // 43
    E_ZCL_ERR_KEY_ESTABLISHMENT_CLUSTER_ENTRY_NOT_FOUND, // 44
    E_ZCL_ERR_KEY_ESTABLISHMENT_CALLBACK_ERROR, // 45
    E_ZCL_ERR_SECURITY_INSUFFICIENT_FOR_CLUSTER, // 46
    E_ZCL_ERR_CUSTOM_COMMAND_HANDLER_NULL_OR_RETURNED_ERROR, // 47
    E_ZCL_ERR_INVALID_IMAGE_SIZE,          // 48
    E_ZCL_ERR_INVALID_IMAGE_VERSION,          // 49
    E_ZCL_READ_ATTR_REQ_NOT_FINISHED,          // 50
    E_ZCL_DENY_ATTRIBUTE_ACCESS,              // 51
    E_ZCL_ERR_SECURITY_FAIL,                  // 52
    E_ZCL_ERR_CLUSTER_COMMAND_NOT_FOUND,
    E_ZCL_ERR_ENUM_END
} teZCL_Status;
```

Table 26. General Return Code Enumerations

Enumeration	Description
E_ZCL_SUCCESS	Function call was successful in its purpose
E_ZCL_FAIL	Function call failed in its purpose and no other error code is appropriate
E_ZCL_ERR_PARAMETER_NULL	Specified parameter pointer was null
E_ZCL_ERR_PARAMETER_RANGE	A parameter value was out-of-range
E_ZCL_ERR_HEAP_FAIL	ZCL heap is out-of-memory
E_ZCL_ERR_EP_RANGE	Specified endpoint number was out-of-range
E_ZCL_ERR_EP_UNKNOWN	Specified endpoint has not been registered with the ZCL (but endpoint number was in-range)
E_ZCL_ERR_SECURITY_RANGE	Security value is out-of-range
E_ZCL_ERR_CLUSTER_0	Specified endpoint has no clusters
E_ZCL_ERR_CLUSTER_NULL	Specified pointer to a cluster was null
E_ZCL_ERR_CLUSTER_NOT_FOUND	Specified cluster has not been registered with the ZCL
E_ZCL_ERR_CLUSTER_ID_RANGE	Specified cluster ID was out-of-range
E_ZCL_ERR_ATTRIBUTES_NULL	Specified pointer to an attribute was null
E_ZCL_ERR_ATTRIBUTES_0	List of attributes to be read was empty
E_ZCL_ERR_ATTRIBUTE_WO	Attempt was made to read write-only attribute
E_ZCL_ERR_ATTRIBUTE_RO	Attempt was made to write to read-only attribute
E_ZCL_ERR_ATTRIBUTES_ACCESS	Error occurred while accessing attribute
E_ZCL_ERR_ATTRIBUTE_TYPE_ UNSUPPORTED	Specified attribute was of unsupported type
E_ZCL_ERR_ATTRIBUTE_NOT_ FOUND	Specified attribute was not found
E_ZCL_ERR_CALLBACK_NULL	Specified pointer to a callback function was null
E_ZCL_ERR_ZBUFFER_FAIL	No buffer available to transmit message

Table 26. General Return Code Enumerations*...continued*

Enumeration	Description
E_ZCL_ERR_ZTRANSMIT_FAIL *	ZigBee PRO stack has reported a transmission error
E_ZCL_ERR_CLIENT_SERVER_ STATUS	Cluster instance of wrong kind (e.g. client instead of server)
E_ZCL_ERR_TIMER_RESOURCE	No timer resource was available
E_ZCL_ERR_ATTRIBUTE_IS_CLIENT	Attempt made by a cluster client to read a client attribute
E_ZCL_ERR_ATTRIBUTE_IS_SERVER	Attempt made by a cluster server to read a server attribute
E_ZCL_ERR_ATTRIBUTE_RANGE	Attribute value is out-of-range
E_ZCL_ERR_ATTRIBUTE_MISMATCH	Reserved for future use
E_ZCL_ERR_KEY_ESTABLISHMENT_ MORE_THAN_ONE_CLUSTER	Attempt made to register more than one Key Establishment cluster on the device (only one is permitted per device)
E_ZCL_ERR_INSUFFICIENT_SPACE	Cluster does not have enough space in its list to store data item, e.g. **eSE_Price AddPriceEntry()** may return this code
E_ZCL_ERR_NO_REPORTABLE_ CHANGE	Reserved for future use
E_ZCL_ERR_NO_REPORT_ENTRIES	Reserved for future use
E_ZCL_ERR_ATTRIBUTE_NOT_ REPORTABLE	Reserved for future use
E_ZCL_ERR_ATTRIBUTE_ID_ORDER **	Attempt made to register a cluster with attribute IDs not defined in ascending order
E_ZCL_ERR_MALFORMED_ MESSAGE	Received ZCL message is not formed correctly. This error code is used in a callback event on the receiving device
E_ZCL_ERR_MANUFACTURER_ SPECIFIC **	Inconsistency in a manufacturer-specific cluster definition has been found
E_ZCL_ERR_PROFILE_ID **	Profile ID of a cluster is not valid - for example, the cluster being registered is not manufacturer-specific but the profile ID is in range reserved for manufacturer-specific profiles
E_ZCL_ERR_INVALID_VALUE	An invalid value has been detected.
E_ZCL_ERR_CERT_NOT_FOUND	Reserved for future use
E_ZCL_ERR_CUSTOM_DATA_NULL	Custom data associated with cluster is NULL
E_ZCL_ERR_TIME_NOT_ SYNCHRONISED	Time has not been synchronized by calling **vZCL_SetUTC-Time()**. This error code is returned by functions that require time to be synchronised, for example, **e SE_PriceAddPriceEntry()**
E_ZCL_ERR_SIGNATURE_VERIFY_ FAILED	Reserved for future use
E_ZCL_ERR_ZRECEIVE_FAIL *	ZigBee PRO stack has reported a receive error
E_ZCL_ERR_KEY_ESTABLISHMENT_ END_POINT_NOT_FOUND	Key Establishment endpoint has not been registered correctly
E_ZCL_ERR_KEY_ESTABLISHMENT_ CLUSTER_ENTRY_NOT_FOUND	Key Establishment cluster has not been registered correctly
E_ZCL_ERR_KEY_ESTABLISHMENT_ CALLBACK_ERROR	Key Establishment cluster callback function has returned an error

Table 26. General Return Code Enumerations*...continued*

Enumeration	Description
E_ZCL_ERR_SECURITY_INSUFFICIENT_FOR_CLUSTER	Cluster that requires application-level (APS) security has been accessed using a packet that has not been encrypted with the application link key
E_ZCL_ERR_CUSTOM_COMMAND_HANDLER_NULL_OR_RETURNED_ERROR	No custom handler has been registered for the command or the custom handler for the command has not returned E_ZCL_SUCCESS
E_ZCL_ERR_INVALID_IMAGE_SIZE	OTA image size is not in the correct range
E_ZCL_ERR_INVALID_IMAGE_VERSION	OTA image version is not in the correct range
E_ZCL_READ_ATTR_REQ_NOT_FINISHED	'Read attributes' request not completely fulfilled
E_ZCL_DENY_ATTRIBUTE_ACCESS	Write access to attribute is denied
E_ZCL_ERR_SECURITY_FAIL	Security failure
E_ZCL_ERR_CLUSTER_COMMAND_NOT_FOUND	The cluster command was not found
E_ZCL_ERR_INVALID_VALUE	Reserved for future use

* ZigBee PRO stack raises an error which can be retrieved using **eZCL_GetLastZpsError()**.

** This error code is returned by **eZCL_Register()**, used in designing custom clusters

7.3 ZCL Event Enumerations

The ZCL event types are enumerated in the `teZCL_CallBackEventType` structure below and described in Table 22. An event must be wrapped in a structure of type `tsZCL_CallBackEvent`, detailed in Section 6.2, with the `eEventType` field set to one of the enumerations in the table. The event must be passed into the ZCL using the function **vZCL_EventHandler()**, detailed in Section 5.1. Event handling is fully described in Chapter 3.

```
typedef enum
{
    E_ZCL_CBET_LOCK_MUTEX = 0x0,
    E_ZCL_CBET_UNLOCK_MUTEX,
    E_ZCL_CBET_UNHANDLED_EVENT,
    E_ZCL_CBET_READ_INDIVIDUAL_ATTRIBUTE_RESPONSE,
    E_ZCL_CBET_READ_ATTRIBUTES_RESPONSE,
    E_ZCL_CBET_READ_REQUEST,
    E_ZCL_CBET_REPORT_REQUEST,
    E_ZCL_CBET_DEFAULT_RESPONSE,
    E_ZCL_CBET_ERROR,
    E_ZCL_CBET_TIMER,
    E_ZCL_CBET_ZIGBEE_EVENT,
    E_ZCL_CBET_CLUSTER_CUSTOM,
    E_ZCL_CBET_WRITE_INDIVIDUAL_ATTRIBUTE,
    E_ZCL_CBET_WRITE_ATTRIBUTES,
    E_ZCL_CBET_WRITE_INDIVIDUAL_ATTRIBUTE_RESPONSE,
    E_ZCL_CBET_WRITE_ATTRIBUTES_RESPONSE,
    E_ZCL_CBET_CHECK_ATTRIBUTE_RANGE,
    E_ZCL_CBET_REPORT_TIMEOUT,
    E_ZCL_CBET_REPORT_INDIVIDUAL_ATTRIBUTE,
    E_ZCL_CBET_REPORT_ATTRIBUTES,
    E_ZCL_CBET_REPORT_INDIVIDUAL_ATTRIBUTES_CONFIGURE_RESPONSE,
    E_ZCL_CBET_REPORT_ATTRIBUTES_CONFIGURE,
```

```
    E_ZCL_CBET_REPORT_INDIVIDUAL_ATTRIBUTES_CONFIGURE,
    E_ZCL_CBET_REPORT_ATTRIBUTES_CONFIGURE_RESPONSE,
    E_ZCL_CBET_REPORT_READ_INDIVIDUAL_ATTRIBUTE_CONFIGURATION_RESPONSE,
    E_ZCL_CBET_REPORT_READ_ATTRIBUTE_CONFIGURATION_RESPONSE,
    E_ZCL_CBET_DISCOVER_INDIVIDUAL_ATTRIBUTE_RESPONSE,
    E_ZCL_CBET_DISCOVER_ATTRIBUTES_RESPONSE,
    E_ZCL_CBET_CLUSTER_UPDATE,
    E_ZCL_CBET_ATTRIBUTE_REPORT_MIRROR,
    E_ZCL_CBET_REPORT_REQUEST,
    E_ZCL_CBET_ENABLE_MS_TIMER,
    E_ZCL_CBET_DISABLE_MS_TIMER,
    E_ZCL_CBET_TIMER_MS,
    E_ZCL_CBET_ZGP_DATA_IND_ERROR,
    E_ZCL_CBET_DISCOVER_INDIVIDUAL_COMMAND_RECEIVED_RESPONSE,
    E_ZCL_CBET_DISCOVER_COMMAND_RECEIVED_RESPONSE,
    E_ZCL_CBET_DISCOVER_INDIVIDUAL_COMMAND_GENERATED_RESPONSE,
    E_ZCL_CBET_DISCOVER_COMMAND_GENERATED_RESPONSE,
    E_ZCL_CBET_DISCOVER_INDIVIDUAL_ATTRIBUTE_EXTENDED_RESPONSE,
    E_ZCL_CBET_DISCOVER_ATTRIBUTES_EXTENDED_RESPONSE,
    E_ZCL_CBET_ENUM_END
} teZCL_CallBackEventType;
```

The above enumerations are described in the table below.

Table 27. ZCL Event Types

Event Type Enumeration	Description
E_ZCL_CBET_LOCK_MUTEX	Indicates that a mutex needs to be locked by the application. This event can be generated only when cooperative tasks are disabled in the compile-time options (see Section 1.3)
E_ZCL_CBET_UNLOCK_MUTEX	Indicates that a mutex needs to be unlocked by the application. This event can be generated only when cooperative tasks are disabled in the compile-time options (see Section 1.3)
E_ZCL_CBET_UNHANDLED_EVENT	Indicates that a stack event has been received that cannot be handled by the ZCL (for example, a Data Confirm)
E_ZCL_CBET_READ_INDIVIDUAL_ATTRIBUTE_RESPONSE	Generated for each attribute included in a ‘read attributes’ response
E_ZCL_CBET_READ_ATTRIBUTES_RESPONSE	Indicates that a ‘read attributes’ response has been received
E_ZCL_CBET_READ_REQUEST	Indicates that a ‘read attributes’ request has been received (giving an opportunity for the local application to update the shared structure before it is read)
E_ZCL_CBET_DEFAULT_RESPONSE	Indicates that a ZCL default response message has been received (which indicates an error or that a command has been processed)
E_ZCL_CBET_ERROR	Indicates that a stack event has been received that can-not be handled by the ZCL
E_ZCL_CBET_TIMER	Indicates that a one-second tick of the real-time clock has occurred or that the ZCL timer has expired
E_ZCL_CBET_ZIGBEE_EVENT	Indicates that a ZigBee PRO stack event has occurred
E_ZCL_CBET_CLUSTER_CUSTOM	Indicates that a custom event which is specific to a cluster has occurred
E_ZCL_CBET_WRITE_INDIVIDUAL_ATTRIBUTE	Indicates that an attempt has been made to write an attribute in the shared structure, following a ‘write attributes’ request, and indicates success or failure

Table 27. ZCL Event Types*...continued*

Event Type Enumeration	Description
E_ZCL_CBET_WRITE_ATTRIBUTES	Indicates that all the relevant attributes have been written in the shared structure, following a 'write attributes' request
E_ZCL_CBET_WRITE_INDIVIDUAL_ ATTRIBUTE_RESPONSE	Generated for each attribute included in a 'write attributes' response (this event contains only those attributes for which the writes have failed)
E_ZCL_CBET_WRITE_ATTRIBUTES_ RESPONSE	Indicates that a 'write attributes' response has been received and has been parsed
E_ZCL_CBET_CHECK_ATTRIBUTE_ RANGE	Generated for each attribute included in a received 'write attributes' request, and prompts the application to perform a range check on the new attribute value and to decide whether a write access to the relevant attribute in the shared structure are allowed or disallowed
E_ZCL_CBET_REPORT_TIMEOUT	Indicates that an attribute report is overdue
E_ZCL_CBET_REPORT_ INDIVIDUAL_ATTRIBUTE	Generated for each attribute included in a received attribute report
E_ZCL_CBET_REPORT_ATTRIBUTES	Indicates that all attributes included in a received attribute report have been parsed
E_ZCL_CBET_REPORT_ INDIVIDUAL_ATTRIBUTES_ CONFIGURE_RESPONSE	Generated for each attribute included in a 'configure attributes' response
E_ZCL_CBET_REPORT_ ATTRIBUTES_CONFIGURE	Indicates that all attributes included in a 'configure reporting' request have been parsed
E_ZCL_CBET_REPORT_ INDIVIDUAL_ATTRIBUTES_ CONFIGURE	Generated for each attribute included in a 'configure reporting' request
E_ZCL_CBET_REPORT_ ATTRIBUTES_CONFIGURE_ RESPONSE	Indicates that all attributes included in a 'configure reporting' response have been reported
E_ZCL_CBET_REPORT_READ_ INDIVIDUAL _ATTRIBUTE_CONFIGURATION_ RESPONSE	Generated for each attribute included in a 'read reporting configuration' response
E_ZCL_CBET_REPORT_READ_ ATTRIBUTE_CONFIGURATION_ RESPONSE	Indicates that all attributes included in a 'read reporting configuration' response have been reported
E_ZCL_CBET_DISCOVER_ INDIVIDUAL_ATTRIBUTE_ RESPONSE	Generated for each attribute included in a 'discover attributes' response
E_ZCL_CBET_DISCOVER_ ATTRIBUTES _RESPONSE	Indicates that all attributes included in a 'discover attributes' response have been reported
E_ZCL_CBET_CLUSTER_UPDATE	Indicates that a cluster attribute value may have been changed on the local device
E_ZCL_CBET_ENABLE_MS_TIMER	Indicates that a millisecond timer needs to be started
E_ZCL_CBET_DISABLE_MS_TIMER	Indicates that a millisecond timer needs to be stopped
E_ZCL_CBET_TIMER_MS	Indicates that a millisecond timer has expired

Table 27. ZCL Event Types*...continued*

Event Type Enumeration	Description
E_ZCL_CBET_ZGP_DATA_IND_ ERROR	Indicates that a ZigBee Green Power data indication error has occurred
E_ZCL_CBET_DISCOVER_ INDIVIDUAL_COMMAND_RECEIVED_ RESPONSE	Generated for each command (that can be received) included in a ‘command discovery‘ response
E_ZCL_CBET_DISCOVER_ COMMAND_RECEIVED_RESPONSE	Indicates that all commands (that can be received) included in a ‘command discovery’ response have been reported
E_ZCL_CBET_DISCOVER_ INDIVIDUAL_COMMAND_ GENERATED_RESPONSE	Generated for each command (that can be generated) included in a ‘command discovery‘ response
E_ZCL_CBET_DISCOVER_ COMMAND_GENERATED_R ESPONSE	Indicates that all commands (that can be generated) included in a ‘command discovery’ response have been reported
E_ZCL_CBET_DISCOVER_ INDIVIDUAL_ATTRIBUTE_ EXTENDED_RESPONSE	Generated for each attribute included in a ‘discover attributes extended’ response
E_ZCL_CBET_DISCOVER_ ATTRIBUTES_EXTENDED_ RESPONSE	Indicates that all attributes included in a ‘discover attributes extended’ response have been reported
E_ZCL_CBET_REPORT_TIMEOUT	Reserved for future use
E_ZCL_CBET_REPORT_ INDIVIDUAL_ATTRIBUTE	Reserved for future use
E_ZCL_CBET_REPORT_ATTRIBUTES	Reserved for future use
E_ZCL_CBET_REPORT_ INDIVIDUAL_ATTRIBUTES_ CONFIGURE_RESPONSE	Reserved for future use
E_ZCL_CBET_REPORT_ ATTRIBUTES_CONFIGURE	Reserved for future use
E_ZCL_CBET_REPORT_ INDIVIDUAL_ATTRIBUTES_ CONFIGURE	Reserved for future use
E_ZCL_CBET_REPORT_ ATTRIBUTES_CONFIGURE_ RESPONSE	Reserved for future use
E_ZCL_CBET_REPORT_READ_ INDIVIDUAL_ATTRIBUTE_ CONFIGURATION_RESPONSE	Reserved for future use
E_ZCL_CBET_REPORT_READ_ ATTRIBUTE_CONFIGURATION_ RESPONSE	Reserved for future use
E_ZCL_CBET_DISCOVER_ INDIVIDUAL_ATTRIBUTE_ RESPONSE	Reserved for future use
E_ZCL_CBET_DISCOVER_ ATTRIBUTES_RESPONSE	Reserved for future use

Note: *The structure* `teZCL_CallBackEventType` *is extended by the EZ-mode Commissioning module with the events listed and described in Section 40.5. These events are only included if this module is used, in which case they are added after E_ZCL_CBET_ENUM_END.*

Part III: General Clusters

This Part, General Clusters, comprises fifteen chapters:

- Chapter 8 details the **Basic** cluster
- Chapter 9 details the **Power Configuration** cluster
- Chapter 10 details the **Device Temperature Configuration** cluster
- Chapter 11 details the **Identify** cluster
- Chapter 12 details the **Groups** cluster
- Chapter 13 details the **Scenes** cluster
- Chapter 14 details the **On/Off** cluster
- Chapter 15 details the **On/Off Switch Configuration** cluster
- Chapter 16 details the **Level Control** cluster
- Chapter 17 details the **Alarms** cluster
- Chapter 18 details the **Time** cluster, as well as the use of ZCL time
- Chapter 19 details the **Input and Output** clusters
- Chapter 20 details the **Poll Control** cluster
- Chapter 21 details the **Power Profile** cluster
- Chapter 22 details the **Diagnostics** cluster

8 Basic Cluster

This chapter details the Basic cluster which is a mandatory cluster for all ZigBee devices.

The Basic cluster has a Cluster ID of 0x0000.

8.1 Overview

All devices implement the Basic cluster as a Server-side (input) cluster, so the cluster is able to store attributes and respond to commands relating to these attributes. The cluster's attributes hold basic information about the node (and apply to devices associated with all active endpoints on the host node). The information that can potentially be stored in this cluster comprises: ZCL version, application version, stack version, hardware version, manufacturer name, model identifier, date, power source.

Note: *The Basic cluster can also be implemented as a Client-side (output) cluster to allow the host device to act as a commissioning tool.*

The Basic cluster contains only two mandatory attributes, the remaining attributes being optional - see Section 8.2.

Note: *Since the Basic cluster contains information about the entire node, only one set of Basic cluster attributes must be stored on the node, even if there are multiple instances of the Basic cluster server across multiple devices/endpoints. All cluster instances must refer to the same structure containing the attribute values.*

The Basic cluster is enabled by defining CLD_BASIC in the **zcl_options.h** file.

A Basic cluster instance can act as a client and/or a server. The inclusion of the client or server software must be pre-defined in the application's compile-time options (in addition, if the cluster is to reside on a custom endpoint then the role of client or server must also be specified when creating the cluster instance).

The compile-time options for the Basic cluster are fully detailed in Section 8.6.

8.2 Basic Cluster structure and attributes

The Basic cluster is contained in the following `tsCLD_Basic` structure:

```
typedef struct
{
#ifdef BASIC_SERVER
    zuint8                 u8ZCLVersion;
#ifdef CLD_BAS_ATTR_APPLICATION_VERSION
    zuint8                 u8ApplicationVersion;
#endif
#ifdef CLD_BAS_ATTR_STACK_VERSION
    zuint8                u8StackVersion;
#endif
#ifdef CLD_BAS_ATTR_HARDWARE_VERSION
    zuint8                u8HardwareVersion;
#endif
#ifdef CLD_BAS_ATTR_MANUFACTURER_NAME
    tsZCL_CharacterString sManufacturerName;
uint8                     au8ManufacturerName[32];
#endif
#ifdef CLD_BAS_ATTR_MODEL_IDENTIFIER
    tsZCL_CharacterString sModelIdentifier;
    uint8                 au8ModelIdentifier[32];
#endif
#ifdef CLD_BAS_ATTR_DATE_CODE
    tsZCL_CharacterString sDateCode;
```

```
    uint8                au8DateCode[16];
#endif
    zenum8               ePowerSource;
#ifdef CLD_BAS_ATTR_GENERIC_DEVICE_CLASS
    zenum8               eGenericDeviceClass;
#endif
#ifdef CLD_BAS_ATTR_GENERIC_DEVICE_TYPE
    zenum8               eGenericDeviceType;
#endif
#ifdef CLD_BAS_ATTR_PRODUCT_CODE
    tsZCL_OctetString    sProductCode;
uint8
au8ProductCode[CLD_BASIC_MAX_NUMBER_OF_BYTES_PRODUCT_CODE];
#endif
#ifdef CLD_BAS_ATTR_PRODUCT_URL
    tsZCL_CharacterString  sProductURL;
    uint8
    au8ProductURL[CLD_BASIC_MAX_NUMBER_OF_BYTES_PRODUCT_URL];
#endif
#ifdef CLD_BAS_ATTR_LOCATION_DESCRIPTION
    tsZCL_CharacterString  sLocationDescription;
    uint8                  au8LocationDescription[16];
#endif
#ifdef CLD_BAS_ATTR_PHYSICAL_ENVIRONMENT
    zenum8                 u8PhysicalEnvironment;
#endif
#ifdef CLD_BAS_ATTR_DEVICE_ENABLED
    zbool                bDeviceEnabled;
#endif
#ifdef CLD_BAS_ATTR_ALARM_MASK
    zbmap8               u8AlarmMask;
#endif
#ifdef CLD_BAS_ATTR_DISABLE_LOCAL_CONFIG
    zbmap8               u8DisableLocalConfig;
#endif
#ifdef CLD_BAS_ATTR_SW_BUILD_ID
    tsZCL_CharacterString sSWBuildID;
    uint8                au8SWBuildID[16];
#endif
#endif
    zuint16             u16ClusterRevision;
} tsCLD_Basic;
```

where:

- `u8ZCLVersion` is an 8-bit version number which represents a published set of foundation items, such as global commands and functional descriptions. Currently this should be set to 2.
- `u8ApplicationVersion` is an optional 8-bit attribute which represents the version of the application (and is manufacturer-specific)
- `u8StackVersion` is an optional 8-bit attribute which represents the version of the ZigBee stack used (and is manufacturer-specific)
- `u8HardwareVersion` is an optional 8-bit attribute which represents the version of the hardware used for the device (and is manufacturer-specific)
- The following optional pair of attributes are used to store the name of the manufacturer of the device:
  - `sManufacturerName` is a `tsZCL_CharacterString` structure (see Section 6.1.14) for a string of up to 32 characters representing the manufacturer's name

  - au8ManufacturerName[32] is a byte-array which contains the character data bytes representing the manufacturer's name
- The following optional pair of attributes are used to store the identifier for the model of the device:
  - sModelIdentifier is a tsZCL_CharacterString structure (see Section 6.1.14) for a string of up to 32 characters representing the model identifier
  - au8ModelIdentifier[32] is a byte-array which contains the character data bytes representing the model identifier
- The following optional pair of attributes are used to store manufacturing information about the device:
  - sDateCode is a tsZCL_CharacterString structure (see Section 6.1.14) for a string of up to 16 characters in which the 8 most significant characters contain the date of manufacture in the format YYYYMMDD and the 8 least significant characters contain manufacturer-defined information such as country of manufacture, factory identifier, production line identifier
  - au8DateCode[16] is a byte-array which contains the character data bytes representing the manufacturing information

Note: *The application device code automatically sets two of the fields of* `sDataCode`*. The field* `sDataCode.pu8Data` *is set to point at* `au8DateCode` *and the field* `sDataCode.u8MaxLength` *is set to 16 (see Section 6.1.14 for details of these fields).*

- ePowerSource 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. Enumerations are provided to cover all possibilities - see Section 8.5.2

Note: *The power source in the Basic cluster is completely unrelated to the Node Power descriptor in the ZigBee PRO stack. The power source in the ZigBee PRO stack is set using the ZPS Configuration Editor.*

- eGenericDeviceClass is an optional attribute that identifies the field of application in which the local device type operates (see eGenericDeviceType below). Enumerations are provided - see Section 8.5.3. Currently, the attribute is used only in lighting applications, for which the value is 0x00 (all other values are reserved).
- eGenericDeviceType is an optional attribute that identifies the local device type. Enumerations are provided to cover the different possibilities - see Section 8.5.4. Currently, the attribute is used only in lighting applications.
- The following optional pair of attributes are used to store a code for the product (this attribute may be used in lighting applications only):
  - sProductCode is a tsZCL_OctetString structure (see Section 6.1.14) for a string representing the product code - the maximum number of characters is defined at compile-time (see Section 8.6) using the macro CLD_BASIC_MAX_NUMBER_OF_BYTES_PRODUCT_CODE.
  - au8ProductCode[] is a byte-array which contains the character data bytes representing the product code - the number of array elements, and therefore characters, is determined at compile-time, as indicated above.
- The following optional pair of attributes are used to store a URL for the product (this attribute may be used in lighting application only):
  - sProductURL is a tsZCL_CharacterString structure (see Section 6.1.14) for a character string representing the product URL - the maximum number of characters is defined at compile-time (see Section 8.6) using the macro CLD_BASIC_MAX_NUMBER_OF_BYTES_PRODUCT_URL.
  - au8ProductURL[] is a byte-array which contains the character data bytes representing the product URL - the number of array elements, and therefore characters, is determined at compile-time, as indicated above.
- The following optional pair of attributes relates to the location of the device:
  - sLocationDescription is a tsZCL_CharacterString structure (see Section 6.1.14) for a string of up to 16 characters representing the location of the device
  - au8LocationDescription[16] is a byte-array which contains the character data bytes representing the location of the device

- `u8PhysicalEnvironment` is an optional 8-bit attribute which indicates the physical environment of the device. Enumerations are provided to cover the different possibilities - see Section 8.5.5.
- `bDeviceEnabled` is an optional Boolean attribute which indicates whether the device is enabled (TRUE) or disabled (FALSE). A disabled device cannot send or respond to application level commands other than commands to read or write attributes
- `u8AlarmMask` is an optional bitmap indicating the general alarms that can be generated (Bit 0 - general software alarm, Bit 1 - general hardware alarm)
- `u8DisableLocalConfig` is an optional bitmap allowing the local user interface of the device to be disabled (Bit 0 - 'Reset to factory defaults' buttons, Bit 1 - 'Device configuration' buttons)
- The following optional pair of attributes are used to store a manufacturer-specific software build identifier:
  - `sSWBuildID` is a `tsZCL_CharacterString` structure (see Section 6.1.14) for a string of up to 16 characters representing the software build identifier
  - `au8SWBuildID[16]` is a byte-array which contains the character data bytes representing the software build identifier.
  - u16ClusterRevision is a mandatory attribute that specifies the revision of the cluster specification on which this cluster instance is based. The cluster specification in the ZCL r6 corresponds to a cluster revision of 1. The value is incremented by one for each subsequent revision of the cluster specification. This attribute is also described in Section 2.4

The Basic cluster structure contains three mandatory elements: `u8ZCLVersion`, `ePowerSource` and `u16ClusterRevision`. The remaining elements are optional, each being enabled/disabled through a corresponding macro defined in the **zcl_options.h** file - for example, the attribute `u8ApplicationVersion` is enabled/disabled using the enumeration CLD_BAS_ATTR_APPLICATION_VERSION (see Section 8.3).

The mandatory attribute settings are described further in Section 8.3.

8.3 Mandatory Attribute Settings

The application must set the values of the mandatory `u8ZCLVersion`, `ePowerSource` and `u16ClusterRevision` 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 - for example, **eZLO_Register DimmableLightEndPoint()**.

These values can be set by calling the **eZCL_WriteLocalAttributeValue()** function with the appropriate input values. Alternatively, they can be set by writing to the relevant members of the shared structure of the device, as illustrated below, where `sLight` or `sSwitch` is the device that is registered using the registration function.

On a Dimmable Light:

```
sLight.sBasicCluster.u8ZCLVersion = 0x01;
sLight.sBasicCluster.ePowerSource = E_CLD_BAS_PS_SINGLE_PHASE_MAINS;
sLight.sBasicClusteru.16ClusterRevision = CLD_BAS_CLUSTER_REVISION;
```

On a battery-powered Dimmer Switch:

```
sSwitch.sLocalBasicCluster.u8ZCLVersion = 0x01;
sSwitch.sLocalBasicCluster.ePowerSource = E_CLD_BAS_PS_BATTERY;
sLight.sBasicClusteru.16ClusterRevision = CLD_BAS_CLUSTER_REVISION;
```

8.4 Functions

The following two Basic cluster functions are provided in the NXP implementation of the ZCL:

- eCLD_BasicCreateBasic
- eCLD_BasicCommandResetToFactoryDefaultsSend

8.4.1 eCLD_BasicCreateBasic

```
teZCL_Status eCLD_BasicCreateBasic(
        tsZCL_ClusterInstance *psClusterInstance,
        bool_t bIsServer,
        tsZCL_ClusterDefinition *psClusterDefinition,
        void *pvEndPointSharedStructPtr,
        uint8 *pu8AttributeControlBits);
```

Description

This function creates an instance of the Basic 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 Basic cluster instance on the endpoint, but instances of other clusters may also be created on the same endpoint by calling their corresponding creation functions.

Note: *This function must not be called for an endpoint on which a standard ZigBee device (for example, a Simple Sensor) is used. In this case, the device and its supported clusters must be registered on the endpoint using the relevant device registration function.*

When used, this function must be the first Basic cluster function called in the application, and must be called after the stack has been started and after the ZCL has been initialized.

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 Basic cluster. The function initializes the array elements to zero.

Parameters

- *psClusterInstance*: Pointer to structure containing information about the cluster instance to be created (see Section 6.1.16). This structure is updated by the function by initializing individual structure fields.
- *bIsServer* : Type of cluster instance (server or client) to be created:
  - TRUE - server
  - FALSE - client
- *psClusterDefinition*: Pointer to structure indicating the type of cluster to be created (see Section 6.1.2). In this case, this structure must contain the details of the Basic cluster. This parameter can refer to a pre-filled structure called `sCLD_Basic` which is provided in the **Basic.h** file.
- *pvEndPointSharedStructPtr*: Pointer to the shared structure used for attribute storage. This parameter should be the address of the structure of type `tsCLD_Basic` which defines the attributes of Basic cluster. The function initializes the attributes with default values.
- *pu8AttributeControlBits*: Pointer to an array of **uint8** values, with one element for each attribute in the cluster (see above).

Returns

E_ZCL_SUCCESS
E_ZCL_ERR_PARAMETER_NULL

8.4.2 eCLD_BasicCommandResetToFactoryDefaultsSend

```
teZCL_Status eCLD_BasicCommandResetToFactoryDefaultsSend(
        uint8 u8SourceEndPointId,
        uint8 u8DestinationEndPointId,
        tsZCL_Address *psDestinationAddress,
        uint8 *pu8TransactionSequenceNumber);
```

Description

This function can be called on a client device to send a 'Reset To Factory Defaults' command, requesting the recipient server device to reset to its factory defaults. The recipient device generates a callback event on the endpoint on which the Basic cluster was registered.

If used, the '`Reset To Factory Defaults`' command must be enabled in the compile-time options on both the client and server, as described in Section 8.6.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is 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 to send the request. This parameter is used both to send the command and to identify the instance of the shared structure holding the required attribute values.
- *u8DestinationEndPointId*: Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP
- *psDestinationAddress*: Pointer to a structure holding the address of the node to which the request is sent
- *pu8TransactionSequenceNumber*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the request

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

8.5 Enumerations

8.5.1 teCLD_BAS_ClusterID

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

```
typedef enum
{
```

```
    E_CLD_BAS_ATTR_ID_ZCL_VERSION = 0x0000, /* Mandatory */
    E_CLD_BAS_ATTR_ID_APPLICATION_VERSION,
    E_CLD_BAS_ATTR_ID_STACK_VERSION,
    E_CLD_BAS_ATTR_ID_HARDWARE_VERSION,
    E_CLD_BAS_ATTR_ID_MANUFACTURER_NAME,
    E_CLD_BAS_ATTR_ID_MODEL_IDENTIFIER,
    E_CLD_BAS_ATTR_ID_DATE_CODE,
    E_CLD_BAS_ATTR_ID_POWER_SOURCE, /* Mandatory */
    E_CLD_BAS_ATTR_ID_LOCATION_DESCRIPTION = 0x0010,
    E_CLD_BAS_ATTR_ID_PHYSICAL_ENVIRONMENT,
    E_CLD_BAS_ATTR_ID_DEVICE_ENABLED,
    E_CLD_BAS_ATTR_ID_ALARM_MASK,
    E_CLD_BAS_ATTR_ID_DISABLE_LOCAL_CONFIG,
    E_CLD_BAS_ATTR_ID_SW_BUILD_ID = 0x4000
} teCLD_BAS_ClusterID;
```

8.5.2 teCLD_BAS_PowerSource

The following enumerations are used in the Basic cluster to specify the power source for a device (see above):

```
typedef enum
{
    E_CLD_BAS_PS_UNKNOWN = 0x00,
    E_CLD_BAS_PS_SINGLE_PHASE_MAINS,
    E_CLD_BAS_PS_THREE_PHASE_MAINS,
    E_CLD_BAS_PS_BATTERY,
    E_CLD_BAS_PS_DC_SOURCE,
    E_CLD_BAS_PS_EMERGENCY_MAINS_CONSTANTLY_POWERED,
    E_CLD_BAS_PS_EMERGENCY_MAINS_AND_TRANSFER_SWITCH,
    E_CLD_BAS_PS_UNKNOWN_BATTERY_BACKED = 0x80,
    E_CLD_BAS_PS_SINGLE_PHASE_MAINS_BATTERY_BACKED,
    E_CLD_BAS_PS_THREE_PHASE_MAINS_BATTERY_BACKED,
    E_CLD_BAS_PS_BATTERY_BATTERY_BACKED,
    E_CLD_BAS_PS_DC_SOURCE_BATTERY_BACKED,
    E_CLD_BAS_PS_EMERGENCY_MAINS_CONSTANTLY_POWERED_BATTERY_BACKED,
    E_CLD_BAS_PS_EMERGENCY_MAINS_AND_TRANSFER_SWITCH_BATTERY_BACKED,
} teCLD_BAS_PowerSource;
```

The power source enumerations are described in the table below.

Table 28. Power Source Enumerations

Enumeration	Description
E_CLD_BAS_PS_UNKNOWN	Unknown power source
E_CLD_BAS_PS_SINGLE_PHASE_MAINS	Single-phase mains powered
E_CLD_BAS_PS_THREE_PHASE_MAINS	Three-phase mains powered
E_CLD_BAS_PS_BATTERY	Battery powered
E_CLD_BAS_PS_DC_SOURCE	DC source
E_CLD_BAS_PS_EMERGENCY_MAINS_CONSTANTLY_POWERED	Constantly powered from emergency mains supply
E_CLD_BAS_PS_EMERGENCY_MAINS_AND_TRANSFER_SWITCH	Powered from emergency mains supply via transfer switch
E_CLD_BAS_PS_UNKNOWN_BATTERY_BACKED	Unknown power source but battery back-up

Table 28. Power Source Enumerations*...continued*

Enumeration	Description
E_CLD_BAS_PS_SINGLE_PHASE_MAINS_BATTERY_BACKED	Single-phase mains powered with battery back-up
E_CLD_BAS_PS_THREE_PHASE_MAINS_BATTERY_BACKED	Three-phase mains powered with battery back-up
E_CLD_BAS_PS_BATTERY_BATTERY_BACKED	Battery powered with battery back-up
E_CLD_BAS_PS_DC_SOURCE_BATTERY_BACKED	DC source with battery back-up
E_CLD_BAS_PS_EMERGENCY_MAINS_CONSTANTLY_POWERED_ BATTERY_BACKED	Constantly powered from emergency mains supply with battery back-up
E_CLD_BAS_PS_EMERGENCY_MAINS_AND_TRANSFER_SWITCH_ BATTERY_BACKED	Powered from emergency mains supply via transfer switch with battery back-up

8.5.3 teCLD_BAS_GenericDeviceClass

The following enumerations are used in the Basic cluster to specify the Device Classes:

```
typedef enum
{
    E_CLD_BAS_GENERIC_DEVICE_CLASS_LIGHTING         = 0x00,
} teCLD_BAS_GenericDeviceClass;
```

8.5.4 eCLD_BAS_GenericDeviceType

The following enumerations are used in the Basic cluster to specify the Device Types:

```
typedef enum
{
    E_CLD_BAS_GENERIC_DEVICE_TYPE_INCANDESCENT       = 0x00,
    E_CLD_BAS_GENERIC_DEVICE_TYPE_SPOTLIGHT_HALOGEN,
    E_CLD_BAS_GENERIC_DEVICE_TYPE_HALOGEN_BULB,
    E_CLD_BAS_GENERIC_DEVICE_TYPE_CFL,
    E_CLD_BAS_GENERIC_DEVICE_TYPE_LINEAR_FLUORESCENT,
    E_CLD_BAS_GENERIC_DEVICE_TYPE_LED_BULB,
    E_CLD_BAS_GENERIC_DEVICE_TYPE_SPOTLIGHT_LED,
    E_CLD_BAS_GENERIC_DEVICE_TYPE_LED_STRIP,
    E_CLD_BAS_GENERIC_DEVICE_TYPE_LED_TUBE,
    E_CLD_BAS_GENERIC_DEVICE_TYPE_INDOOR_LUMINAIRE,
    E_CLD_BAS_GENERIC_DEVICE_TYPE_OUTDOOR_LUMINAIRE,
    E_CLD_BAS_GENERIC_DEVICE_TYPE_PENDANT_LUMINAIRE,
    E_CLD_BAS_GENERIC_DEVICE_TYPE_FLOOR_STANDING_LUMINAIRE,
    E_CLD_BAS_GENERIC_DEVICE_TYPE_CONTROLLER         = 0xE0,
    E_CLD_BAS_GENERIC_DEVICE_TYPE_WALL_SWITCH,
    E_CLD_BAS_GENERIC_DEVICE_TYPE_PORTABLE_REMOTE_CONTROLLER,
    E_CLD_BAS_GENERIC_DEVICE_TYPE_MOTION_OR_LIGHT_SENSOR,
    E_CLD_BAS_GENERIC_DEVICE_TYPE_ACTUATOR          = 0xF0,
    E_CLD_BAS_GENERIC_DEVICE_TYPE_WALL_SOCKET,
    E_CLD_BAS_GENERIC_DEVICE_TYPE_GATEWAY_OR_BRIDGE,
    E_CLD_BAS_GENERIC_DEVICE_TYPE_PLUG_IN_UNIT,
    E_CLD_BAS_GENERIC_DEVICE_TYPE_RETROFIT_ACTUATOR,
    E_CLD_BAS_GENERIC_DEVICE_TYPE_UNSPECIFIED     = 0xFF
} teCLD_BAS_GenericDeviceType;
```

8.5.5 teCLD_BAS_PhysicalEnvironment

The following enumerations are used in the Basic cluster to specify the Physical Environment:

```
typedef enum
{
    E_CLD_BAS_PE_UNSPECIFIED            = 0x00,
    E_CLD_BAS_PE_MIRROR,
    E_CLD_BAS_PE_ATRIUM,
    E_CLD_BAS_PE_BAR,
    E_CLD_BAS_PE_COURTYARD,
    E_CLD_BAS_PE_BATHROOM,
    E_CLD_BAS_PE_BEDROOM,
    E_CLD_BAS_PE_BILLIARD_ROOM,
    E_CLD_BAS_PE_UTILITY_ROOM,
    E_CLD_BAS_PE_CELLAR,
    E_CLD_BAS_PE_STORAGE_CLOSET,
    E_CLD_BAS_PE_THREATER,
    E_CLD_BAS_PE_OFFICE_0x0B,
    E_CLD_BAS_PE_DECK,
    E_CLD_BAS_PE_DEN,
    E_CLD_BAS_PE_DINNING_ROOM,
    E_CLD_BAS_PE_ELECTRICAL_ROOM,
    E_CLD_BAS_PE_ELEVATOR,
    E_CLD_BAS_PE_ENTRY,
    E_CLD_BAS_PE_FAMILY_ROOM,
    E_CLD_BAS_PE_MAIN_FLOOR,
    E_CLD_BAS_PE_UPSTAIRS,
    E_CLD_BAS_PE_DOWNSTAIRS,
    E_CLD_BAS_PE_BASEMENT_LOWER_LEVEL,
    E_CLD_BAS_PE_GALLERY,
    E_CLD_BAS_PE_GAME_ROOM,
    E_CLD_BAS_PE_GARAGE,
    E_CLD_BAS_PE_GYM,
    E_CLD_BAS_PE_HALLWAY,
    E_CLD_BAS_PE_HOUSE,
    E_CLD_BAS_PE_KITCHEN,
    E_CLD_BAS_PE_LAUNDRY_ROOM,
    E_CLD_BAS_PE_LIBRARY,
    E_CLD_BAS_PE_MASTER_BEDROOM,
    E_CLD_BAS_PE_MUD_ROOM,
    E_CLD_BAS_PE_NURSERY,
    E_CLD_BAS_PE_PANTRY,
    E_CLD_BAS_PE_OFFICE_0X24,
    E_CLD_BAS_PE_OUTSIDE,
    E_CLD_BAS_PE_POOL,
    E_CLD_BAS_PE_PORCH,
    E_CLD_BAS_PE_SEWING_ROOM,
    E_CLD_BAS_PE_SITTING_ROOM,
    E_CLD_BAS_PE_STAIRWAY,
    E_CLD_BAS_PE_YARD,
    E_CLD_BAS_PE_ATTIC,
    E_CLD_BAS_PE_HOT_TUB,
    E_CLD_BAS_PE_LIVING_ROOM_0X2E,
    E_CLD_BAS_PE_SAUNA,
    E_CLD_BAS_PE_SHOP_WORKSHOP,
    E_CLD_BAS_PE_GUEST_BEDROOM,
    E_CLD_BAS_PE_GUEST_BATH,
    E_CLD_BAS_PE_POWDER_ROOM,
    E_CLD_BAS_PE_BACK_YARD,
```

```
    E_CLD_BAS_PE_FRONT_YARD,
    E_CLD_BAS_PE_PATIO,
    E_CLD_BAS_PE_DRIVEWAY,
    E_CLD_BAS_PE_SUN_ROOM,
    E_CLD_BAS_PE_LIVING_ROOM_0X39,
    E_CLD_BAS_PE_SPA,
    E_CLD_BAS_PE_WHIRLPOOL,
    E_CLD_BAS_PE_SHED,
    E_CLD_BAS_PE_EQUIPMENT_STORAGE,
    E_CLD_BAS_PE_HOBBY_CRAFT_ROOM,
    E_CLD_BAS_PE_FOUNTAIN,
    E_CLD_BAS_PE_POND,
    E_CLD_BAS_PE_RECEPTION_ROOM,
    E_CLD_BAS_PE_BREAKFAST_ROOM,
    E_CLD_BAS_PE_NOOK,
    E_CLD_BAS_PE_GARDEN,
    E_CLD_BAS_PE_PANIC_ROOM,
    E_CLD_BAS_PE_TERRACE,
    E_CLD_BAS_PE_ROOF,
    E_CLD_BAS_PE_TOILET,
    E_CLD_BAS_PE_TOILET_MAIN,
    E_CLD_BAS_PE_OUTSIDE_TOILET,
    E_CLD_BAS_PE_SHOWER_ROOM,
    E_CLD_BAS_PE_STUDY,
    E_CLD_BAS_PE_FRONT_GARDEN,
    E_CLD_BAS_PE_BACK_GARDEN,
    E_CLD_BAS_PE_KETTLE,
    E_CLD_BAS_PE_TELEVISION,
    E_CLD_BAS_PE_STOVE,
    E_CLD_BAS_PE_MICROWAVE,
    E_CLD_BAS_PE_TOASTER,
    E_CLD_BAS_PE_VACUMM,
    E_CLD_BAS_PE_APPLIANCE,
    E_CLD_BAS_PE_FRONT_DOOR,
    E_CLD_BAS_PE_BACK_DOOR,
    E_CLD_BAS_PE_FRIDGE_DOOR,
    E_CLD_BAS_PE_MEDICATION_CABINET_DOOR,
    E_CLD_BAS_PE_WARDROBE_DOOR,
    E_CLD_BAS_PE_FRONT_CUPBOARD_DOOR,
    E_CLD_BAS_PE_OTHER_DOOR,
    E_CLD_BAS_PE_WAITING_ROOM,
    E_CLD_BAS_PE_TRIAGE_ROOM,
    E_CLD_BAS_PE_DOCTOR_OFFICE,
    E_CLD_BAS_PE_PATIENT_PRIVATE_ROOM,
    E_CLD_BAS_PE_CONSULTATION_ROOM,
    E_CLD_BAS_PE_NURSE_STATION,
    E_CLD_BAS_PE_WARD,
    E_CLD_BAS_PE_COORIDOR,
    E_CLD_BAS_PE_OPERATING_THREATER,
    E_CLD_BAS_PE_DENTAL_SURGERY_ROOM,
    E_CLD_BAS_PE_MEDICAL_IMAGING_ROOM,
    E_CLD_BAS_PE_DECONTAMINATION_ROOM,
    E_CLD_BAS_PE_UNKNOWN_ENVIRONMENT
} teCLD_BAS_PhysicalEnvironment;
```

8.6 Compile-time options

To enable the Basic cluster in the code to be built, it is necessary to add the following to the **zcl_options.h** file:

```
#define CLD_BASIC
```

In addition, to include the software for a cluster client or server or both, it is necessary to add one or both of the following to the same file:

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

The Basic cluster contains macros that may be optionally specified at compile-time by adding some or all of the following lines to the **zcl_options.h** file.

Optional Attributes

Add this line to enable the optional Application Version attribute:

```
#define   CLD_BAS_ATTR_APPLICATION_VERSION
```

Add this line to enable the optional Stack Version attribute:

```
#define   CLD_BAS_ATTR_STACK_VERSION
```

Add this line to enable the optional Hardware Version attribute:

```
#define   CLD_BAS_ATTR_HARDWARE_VERSION
```

Add this line to enable the optional Manufacturer Name attribute:

```
#define   CLD_BAS_ATTR_MANUFACTURER_NAME
```

Add this line to enable the optional Model Identifier attribute:

```
#define   CLD_BAS_ATTR_MODEL_IDENTIFIER
```

Add this line to enable the optional Date Code attribute:

```
#define   CLD_BAS_ATTR_DATE_CODE
```

Add this line to enable the optional Generic Class Device attribute:

```
#define CLD_BAS_ATTR_GENERIC_DEVICE_CLASS
```

Add this line to enable the optional Generic Device Type attribute:

```
#define CLD_BAS_ATTR_GENERIC_DEVICE_TYPE
```

Add this line to enable the optional Product Code attribute:

```
#define CLD_BAS_ATTR_PRODUCT_CODE
```

Add this line to enable the optional Product URL attribute:

```
#define CLD_BAS_ATTR_PRODUCT_URL
```

Add this line to enable the optional Location Description attribute:

```
#define   CLD_BAS_ATTR_LOCATION_DESCRIPTION
```

Add this line to enable the optional Physical Environment attribute:

```
#define   CLD_BAS_ATTR_PHYSICAL_ENVIRONMENT
```

Add this line to enable the optional Device Enabled attribute:

```
#define   CLD_BAS_ATTR_DEVICE_ENABLED
```

Add this line to enable the optional Alarm Mask attribute:

```
#define   CLD_BAS_ATTR_ALARM_MASK
```

Add this line to enable the optional Disable Local Config attribute:

```
#define   CLD_BAS_ATTR_DISABLE_LOCAL_CONFIG
```

Add this line to enable the optional Software Build ID attribute:

```
#define   CLD_BAS_ATTR_SW_BUILD_ID
```

Global Attributes

Add this line to define the value (n) of the Cluster Revision attribute:

```
#define CLD_BAS_CLUSTER_REVISION <n>
```

The default value is 1, which corresponds to the revision of the cluster in the ZCL r6 specification (see Section 2.4).

Optional Commands

Add this line to enable the optional Reset To Factory Defaults command on the client and server:

```
#define CLD_BAS_CMD_RESET_TO_FACTORY_DEFAULTS
```

Product Code Length

The default length of the product code contained in the attributes `sProductCode` and `au8ProductCode[]` can be defined by adding the following line:

```
#define CLD_BAS_PCODE_SIZE <n>
```

where `<n>` is the default number characters in the product code.

The maximum length of the product code contained in the attributes `sProductCode` and `au8ProductCode[]` can be defined by adding the following line:

```
#define CLD_BASIC_MAX_NUMBER_OF_BYTES_PRODUCT_CODE <n>
```

where `<n>` is the maximum number characters in the product code.

Product URL Length

The default length of the product URL contained in the attributes `sProductURL` and `au8ProductURL[]` can be defined by adding the following line:

```
#define CLD_BAS_URL_SIZE <n>
```

where `<n>` is the default number characters in the product URL.

The maximum length of the product URL contained in the attributes `sProductURL` and `au8ProductURL[]` can be defined by adding the following line:

```
#define CLD_BASIC_MAX_NUMBER_OF_BYTES_PRODUCT_URL <n>
```

where `<n>` is the maximum number characters in the product URL.

9 Power Configuration Cluster

This chapter describes the Power Configuration cluster which is concerned with the power source(s) of a device.

The Power Configuration cluster has a Cluster ID of 0x0001.

9.1 Overview

The Power Configuration cluster allows:

- information to be obtained about the power source(s) of a device
- voltage alarms to be configured

To use the functionality of this cluster, you must include the file **PowerConfiguration.h** in your application and enable the cluster by defining CLD_POWER_CONFIGURATION in the **zcl_options.h** file.

It is also necessary to enable the cluster as a server or client, or as both:

- The cluster server is able to receive commands to change the power configuration on the local device.
- The cluster client is able to send commands to change the power configuration on the remote device.

The inclusion of the client or server software must be pre-defined in the application's compile-time options (in addition, if the cluster is to reside on a custom endpoint then the role of client or server must also be specified when creating the cluster instance).

The compile-time options for the Power Configuration cluster are fully detailed in Section 9.6.

9.2 Power Configuration Cluster structure and attributes

The structure definition for the Power Configuration cluster is:

```
typedef struct
{
#ifdef POWER_CONFIGURATION_SERVER
#ifdef CLD_PWRCFG_ATTR_MAINS_VOLTAGE
    zuint16  u16MainsVoltage;
#endif
#ifdef CLD_PWRCFG_ATTR_MAINS_FREQUENCY
    zuint8   u8MainsFrequency;
#endif
#ifdef CLD_PWRCFG_ATTR_MAINS_ALARM_MASK
    zbmap8   u8MainsAlarmMask;
#endif
#ifdef CLD_PWRCFG_ATTR_MAINS_VOLTAGE_MIN_THRESHOLD
    uint16   u16MainsVoltageMinThreshold;
#endif
#ifdef CLD_PWRCFG_ATTR_MAINS_VOLTAGE_MAX_THRESHOLD
    uint16   u16MainsVoltageMaxThreshold;
#endif
#ifdef CLD_PWRCFG_ATTR_MAINS_VOLTAGE_DWELL_TRIP_POINT
    uint16   u16MainsVoltageDwellTripPoint;
#endif
#ifdef CLD_PWRCFG_ATTR_BATTERY_VOLTAGE
    uint8    u8BatteryVoltage;
#endif
#ifdef CLD_PWRCFG_ATTR_BATTERY_PERCENTAGE_REMAINING
    uint8    u8BatteryPercentageRemaining;
#endif
```

```
#ifdef CLD_PWRCFG_ATTR_BATTERY_MANUFACTURER
    tsZCL_CharacterString   sBatteryManufacturer;
    uint8    au8BatteryManufacturer[16];
#endif
#ifdef CLD_PWRCFG_ATTR_BATTERY_SIZE
    zenum8   u8BatterySize;
#endif
#ifdef CLD_PWRCFG_ATTR_BATTERY_AHR_RATING
    zuint16      u16BatteryAHRating;
#endif
#ifdef CLD_PWRCFG_ATTR_BATTERY_QUANTITY
    zuint8   u8BatteryQuantity;
#endif
#ifdef CLD_PWRCFG_ATTR_BATTERY_RATED_VOLTAGE
    zuint8   u8BatteryRatedVoltage;
#endif
#ifdef CLD_PWRCFG_ATTR_BATTERY_ALARM_MASK
    zbmap8   u8BatteryAlarmMask;
#endif
#ifdef CLD_PWRCFG_ATTR_BATTERY_VOLTAGE_MIN_THRESHOLD
    zuint8   u8BatteryVoltageMinThreshold;
#endif
#ifdef CLD_PWRCFG_ATTR_ID_BATTERY_VOLTAGE_THRESHOLD1
    zuint8   u8BatteryVoltageThreshold1;
#endif
#ifdef CLD_PWRCFG_ATTR_ID_BATTERY_VOLTAGE_THRESHOLD2
    zuint8   u8BatteryVoltageThreshold2;
#endif
#ifdef CLD_PWRCFG_ATTR_ID_BATTERY_VOLTAGE_THRESHOLD3
    zuint8   u8BatteryVoltageThreshold3;
#endif
#ifdef CLD_PWRCFG_ATTR_ID_BATTERY_PERCENTAGE_MIN_THRESHOLD
    zuint8   u8BatteryPercentageMinThreshold;
#endif
#ifdef CLD_PWRCFG_ATTR_ID_BATTERY_PERCENTAGE_THRESHOLD1
    zuint8   u8BatteryPercentageThreshold1;
#endif
#ifdef CLD_PWRCFG_ATTR_ID_BATTERY_PERCENTAGE_THRESHOLD2
    zuint8   u8BatteryPercentageThreshold2;
#endif
#ifdef CLD_PWRCFG_ATTR_ID_BATTERY_PERCENTAGE_THRESHOLD3
    zuint8   u8BatteryPercentageThreshold3;
#endif
#ifdef CLD_PWRCFG_ATTR_ID_BATTERY_ALARM_STATE
    zbmap32      u32BatteryAlarmState;
#endif
#ifdef CLD_PWRCFG_ATTR_BATTERY_2_VOLTAGE
    uint8    u8Battery2Voltage;
#endif
#ifdef CLD_PWRCFG_ATTR_BATTERY_2_PERCENTAGE_REMAINING
    uint8    u8Battery2PercentageRemaining;
#endif
#ifdef CLD_PWRCFG_ATTR_BATTERY_2_MANUFACTURER
    tsZCL_CharacterString   sBattery2Manufacturer;
    uint8    au8Battery2Manufacturer[16];
#endif
#ifdef CLD_PWRCFG_ATTR_BATTERY_2_SIZE
    zenum8   u8Battery2Size;
#endif
#ifdef CLD_PWRCFG_ATTR_BATTERY_2_AHR_RATING
```

```
    zuint16      u16Battery2AHRating;
#endif
#ifdef CLD_PWRCFG_ATTR_BATTERY_2_QUANTITY
    zuint8   u8Battery2Quantity;
#endif
#ifdef CLD_PWRCFG_ATTR_BATTERY_2_RATED_VOLTAGE
    zuint8   u8Battery2RatedVoltage;
#endif
#ifdef CLD_PWRCFG_ATTR_BATTERY_2_ALARM_MASK
    zbmap8   u8Battery2AlarmMask;
#endif
#ifdef CLD_PWRCFG_ATTR_BATTERY_2_VOLTAGE_MIN_THRESHOLD
    zuint8   u8Battery2VoltageMinThreshold;
#endif
#ifdef CLD_PWRCFG_ATTR_ID_BATTERY_2_VOLTAGE_THRESHOLD1
    zuint8   u8Battery2VoltageThreshold1;
#endif
#ifdef CLD_PWRCFG_ATTR_ID_BATTERY_2_VOLTAGE_THRESHOLD2
    zuint8   u8Battery2VoltageThreshold2;
#endif
#ifdef CLD_PWRCFG_ATTR_ID_BATTERY_2_VOLTAGE_THRESHOLD3
    zuint8   u8Battery2VoltageThreshold3;
#endif
#ifdef CLD_PWRCFG_ATTR_ID_BATTERY_2_PERCENTAGE_MIN_THRESHOLD
    zuint8   u8Battery2PercentageMinThreshold;
#endif
#ifdef CLD_PWRCFG_ATTR_ID_BATTERY_2_PERCENTAGE_THRESHOLD1
    zuint8   u8Battery2PercentageThreshold1;
#endif
#ifdef CLD_PWRCFG_ATTR_ID_BATTERY_2_PERCENTAGE_THRESHOLD2
    zuint8   u8Battery2PercentageThreshold2;
#endif
#ifdef CLD_PWRCFG_ATTR_ID_BATTERY_2_PERCENTAGE_THRESHOLD3
    zuint8   u8Battery2PercentageThreshold3;
#endif
#ifdef CLD_PWRCFG_ATTR_ID_BATTERY_2_ALARM_STATE
    zbmap32      u32Battery2AlarmState;
#endif
#ifdef CLD_PWRCFG_ATTR_BATTERY_3_VOLTAGE
    uint8    u8Battery3Voltage;
#endif
#ifdef CLD_PWRCFG_ATTR_BATTERY_3_PERCENTAGE_REMAINING
    uint8    u8Battery3PercentageRemaining;
#endif
#ifdef CLD_PWRCFG_ATTR_BATTERY_3_MANUFACTURER
    tsZCL_CharacterString   sBattery3Manufacturer;
    uint8    au8Battery3Manufacturer[16];
#endif
#ifdef CLD_PWRCFG_ATTR_BATTERY_3_SIZE
    zenum8   u8Battery3Size;
#endif
#ifdef CLD_PWRCFG_ATTR_BATTERY_3_AHR_RATING
    zuint16      u16Battery3AHRating;
#endif
#ifdef CLD_PWRCFG_ATTR_BATTERY_3_QUANTITY
    zuint8   u8Battery3Quantity;
#endif
#ifdef CLD_PWRCFG_ATTR_BATTERY_3_RATED_VOLTAGE
    zuint8   u8Battery3RatedVoltage;
#endif
```

```
#ifdef CLD_PWRCFG_ATTR_BATTERY_3_ALARM_MASK
    zbmap8   u8Battery3AlarmMask;
#endif
#ifdef CLD_PWRCFG_ATTR_BATTERY_3_VOLTAGE_MIN_THRESHOLD
    zuint8   u8Battery3VoltageMinThreshold;
#endif
#ifdef CLD_PWRCFG_ATTR_ID_BATTERY_3_VOLTAGE_THRESHOLD1
    zuint8   u8Battery3VoltageThreshold1;
#endif
#ifdef CLD_PWRCFG_ATTR_ID_BATTERY_3_VOLTAGE_THRESHOLD2
    zuint8   u8Battery3VoltageThreshold2;
#endif
#ifdef CLD_PWRCFG_ATTR_ID_BATTERY_3_VOLTAGE_THRESHOLD3
    zuint8   u8Battery3VoltageThreshold3;
#endif
#ifdef CLD_PWRCFG_ATTR_ID_BATTERY_3_PERCENTAGE_MIN_THRESHOLD
    zuint8   u8Battery3PercentageMinThreshold;
#endif
#ifdef CLD_PWRCFG_ATTR_ID_BATTERY_3_PERCENTAGE_THRESHOLD1
    zuint8   u8Battery3PercentageThreshold1;
#endif
#ifdef CLD_PWRCFG_ATTR_ID_BATTERY_3_PERCENTAGE_THRESHOLD2
    zuint8   u8Battery3PercentageThreshold2;
#endif
#ifdef CLD_PWRCFG_ATTR_ID_BATTERY_3_PERCENTAGE_THRESHOLD3
    zuint8   u8Battery3PercentageThreshold3;
#endif
#ifdef CLD_PWRCFG_ATTR_ID_BATTERY_3_ALARM_STATE
    zbmap32      u32Battery3AlarmState;
#endif
#ifdef  CLD_PWRCFG_ATTR_ID_ATTRIBUTE_REPORTING_STATUS
    zuint8   u8AttributeReportingStatus;
#endif
#endif
    zuint16      u16ClusterRevision;
} tsCLD_PowerConfiguration;
```

The attributes are classified into four attribute sets: Mains Information, Mains Settings, Battery Information, Battery Settings and Global. The attributes from these sets are described below.

Mains Information Attribute Set

- `u16MainsVoltage` is the measured AC (RMS) mains voltage or DC voltage currently applied to the device, in units of 100 mV.
- `u8MainsFrequency` is half of the measured AC mains frequency, in Hertz, currently applied to the device. Actual frequency = 2 x `u8MainsFrequency`. This allows AC mains frequencies to be stored in the range 2-506 Hz in steps of 2 Hz. In addition:
  - 0x00 indicates a DC supply or that AC frequency is too low to be measured
  - 0xFE indicates that AC frequency is too high to be measured
  - 0xFF indicates that AC frequency could not be measured.

Mains Settings Attribute Set

- `u8MainsAlarmMask` is a bitmap indicating which mains voltage alarms can be generated (a bit is set to ‘1’ if the alarm is enabled):

Bit	Description
0	Under-voltage alarm (triggered when measured RMS mains voltage falls below a pre-defined threshold - see below)
1	Over-voltage alarm (triggered when measured RMS mains voltage rises above a pre-defined threshold - see below)
2	Mains power supply has been lost or is unavailable - that is, the device is now running on battery power.
3-7	Reserved

- `u16MainsVoltageMinThreshold` is the threshold for the under-voltage alarm, in units of 100 mV. The RMS mains voltage is allowed to fall below this threshold for the duration specified by `16MainsVoltageDwellTripPoint` before the alarm is triggered (see below). 0xFFFF indicates that the alarm will not be generated.
- `u16MainsVoltageMaxThreshold` is the threshold for the over-voltage alarm, in units of 100 mV. The RMS mains voltage is allowed to rise above this threshold for the duration specified by `16MainsVoltageDwellTripPoint` before the alarm is triggered (see below). 0xFFFF indicates that the alarm will not be generated.
- `u16MainsVoltageDwellTripPoint` defines the time-delay, in seconds, before an over-voltage or under-voltage alarm will be triggered when the mains voltage crosses the relevant threshold. If the mains voltage returns within the limits of the thresholds during this time, the alarm will be cancelled. 0xFFFF indicates that the alarms will not be generated.

Battery Information Attribute Set (Battery 1)

- `u8BatteryVoltage` is the measured battery voltage currently applied to the device, in units of 100 mV. 0xFF indicates that the measured voltage is invalid or unknown.
- `u8BatteryPercentageRemaining` indicates the remaining battery life as a percentage of the complete battery lifespan, expressed to the nearest half-percent in the range 0 to 100 - for example, 0xAF represents 87.5%. The special value 0xFF indicates an invalid or unknown measurement.

Battery Settings Attribute Set (Battery 1)

- `sBatteryManufacturer` is a pointer to the array containing the name of the battery manufacturer (see below).
- `au8BatteryManufacturer[16]` is a 16-element array containing the name of the battery manufacturer (maximum of 16 characters).
- `u8BatterySize` is an enumeration indicating the type of battery in the device - the enumerations are listed in Section 9.5.2.
- `u16BatteryAHRating` is the Ampere-hour (Ah) charge rating of the battery, in units of 10 mAh.
- `u8BatteryQuantity` is the number of batteries used to power the device.
- `u8BatteryRatedVoltage` is the rated voltage of the battery, in units of 100 mV.
- `u8BatteryAlarmMask` is a bitmap indicating whether the battery-low alarm can be generated - if enabled, the alarm is generated when the battery voltage falls below a pre-defined threshold (see below). The alarm-enable bit is bit 0 (which is set to ‘1’ if the alarm is enabled).
- `u8BatteryVoltageMinThreshold` is the battery voltage threshold, in units of 100 mV, below which the device cannot operate or transmit - a battery-low alarm can be triggered when the battery voltage falls below this threshold:

Value	Description
0x00 - 0x39	Minimum battery voltage threshold, in units of 100 mV
0x3A	Mains power supply has been lost or is unavailable - that is, the device is now running on battery power.
0x3B - 0xFF	Reserved

- `u8BatteryVoltageThreshold1` is a battery voltage threshold, in units of 100 mV, which can correspond to a battery-low alarm - that is, if the battery voltage falls below this threshold, an alarm can be triggered. It must be greater than the value defined for u8BatteryVoltageMinThreshold. The special value 0xFF indicates that the threshold is not used.u8BatteryVoltageThreshold2 is a battery voltage threshold, in units of 100 mV, which can correspond to a battery-low alarm - that is, if the battery voltage falls below this threshold, an alarm can be triggered. It must be greater than the value defined for u8BatteryVoltageThreshold1. The special value 0xFF indicates that the threshold is not used.
- `u8BatteryVoltageThreshold3` is a battery voltage threshold, in units of 100 mV, which can correspond to a battery-low alarm - that is, if the battery voltage falls below this threshold, an alarm can be triggered. It must be greater than the value defined for u8BatteryVoltageThreshold2. The special value 0xFF indicates that the threshold is not used.
- `u8BatteryPercentageMinThreshold` is the minimum alarm threshold for percentage battery-life, expressed in half-percent steps in the range 0 to 100 - if the remaining percentage battery-life (u8BatteryPercentageRemaining) falls below this threshold, an alarm can be triggered.
- `u8BatteryPercentageThreshold1` is an alarm threshold for percentage battery-life, expressed in half-percent steps in the range 0 to 100 - if the remaining percentage battery-life (u8BatteryPercentageRemaining) falls below this threshold, an alarm can be triggered. It must be greater than the value defined for u8BatteryPercentageMinThreshold. The special value 0xFF indicates that the threshold is not used.
- `u8BatteryPercentageThreshold2` is an alarm threshold for percentage battery-life, expressed in half-percent steps in the range 0 to 100 - if the remaining percentage battery-life (u8BatteryPercentageRemaining) falls below this threshold, an alarm can be triggered. It must be greater than the value defined for u8BatteryPercentageThreshold1. The special value 0xFF indicates that the threshold is not used.
- `u8BatteryPercentageThreshold3` is an alarm threshold for percentage battery-life, expressed in half-percent steps in the range 0 to 100 - if the remaining percentage battery-life (u8BatteryPercentageRemaining) falls below this threshold, an alarm can be triggered. It must be greater than the value defined for u8BatteryPercentageThreshold2. The special value 0xFF indicates that the threshold is not used.
- `u32BatteryAlarmState` is a bitmap representing the current state of the alarms for the battery or batteries (the bitmap includes status bits for optional additional batteries 2 and 3). It indicates the state of the battery in relation to the voltage and percentage-life thresholds defined by the attributes above (a bit is set to ‘1’ when the corresponding threshold has been reached).

Bit	Description
Bits for Battery	

0	Bit is set if one of the following thresholds has been reached:
	`u8BatteryVoltageMinThreshold`
	`u8BatteryPercentageMinThreshold`

1	Bit is set if one of the following thresholds has been reached:
	`u8BatteryVoltageThreshold1`
	`u8BatteryPercentageThreshold1`

2	Bit is set if one of the following thresholds has been reached: `u8BatteryVoltageThreshold2`

Bit	Description
	`u8BatteryPercentageThreshold2`

3	Bit is set if one of the following thresholds has been reached:
	`u8BatteryVoltageThreshold3`
	`u8BatteryPercentageThreshold3`

4 - 9	Reserved
Bits for Battery 2 (Optional)	

10	Bit is set if one of the following thresholds has been reached:
	`u8Battery2VoltageMinThreshold`
	`u8Battery2PercentageMinThreshold`

11	Bit is set if one of the following thresholds has been reached:
	`u8Battery2VoltageThreshold1`
	`u8Battery2PercentageThreshold1`

12	Bit is set if one of the following thresholds has been reached:
	`u8Battery2VoltageThreshold2`
	`u8Battery2PercentageThreshold2`

13	Bit is set if one of the following thresholds has been reached:
	`u8Battery2VoltageThreshold3`
	`u8Battery2PercentageThreshold3`

14 - 19	Reserved
Bits for Battery 3 (Optional)	

20	Bit is set if one of the following thresholds has been reached:
	`u8Battery3VoltageMinThreshold`
	`u8Battery3PercentageMinThreshold`

21	Bit is set if one of the following thresholds has been reached:
	`u8Battery3VoltageThreshold1`
	`u8Battery3PercentageThreshold1`

22	Bit is set if one of the following thresholds has been reached:
	`u8Battery3VoltageThreshold2`
	`u8Battery3PercentageThreshold2`

23	Bit is set if one of the following thresholds has been reached:
	`u8Battery3VoltageThreshold3`
	`u8Battery3PercentageThreshold3`

24 - 29	Reserved
30	Mains power supply has been lost or is unavailable - that is, the device is now running on battery power
31	Reserved

Battery Information and Battery Settings Attribute Sets for Battery <X>

The Battery Information and Battery Settings attribute sets are repeated for up to two further (optional) batteries, denoted 2 and 3. The attributes are as follows, where <X> is 2 or 3, and their definitions are identical to those of the equivalent attributes in the Battery Information and Battery Settings attribute sets described above.

```
u8Battery<X>Voltage
u8Battery<X>PercentageRemaining
au8Battery<X>Manufacturer[16]
u8Battery<X>Size
u16Battery<X>AHRating
u8Battery<X>Quantity
u8Battery<X>RatedVoltage
u8Battery<X>AlarmMask
u8Battery<X>VoltageMinThreshold
u8Battery<X>VoltageThreshold1
u8Battery<X>VoltageThreshold2
u8Battery<X>VoltageThreshold3
u8Battery<X>PercentageMinThreshold
u8Battery<X>PercentageThreshold1
u8Battery<X>PercentageThreshold2
u8Battery<X>PercentageThreshold3
u32Battery<X>AlarmState
```

Global Attribute Set

```
u8AttributeReportingStatus is an optional attribute that should be enabled when attribute
u16ClusterRevision is a mandatory attribute that specifies the revision of the cluster spe
```

9.3 Attributes for Default Reporting

The following attributes of the Power Configuration cluster can be selected for default reporting:

```
u8BatteryPercentageRemaining
u32BatteryAlarmState
```

Attribute reporting (including default reporting) is described in Appendix B. Enabling reports for these attributes is described in Appendix B.3.6.

9.4 Functions

The below Power Configuration cluster function is provided in the NXP implementation of the ZCL:

eCLD_PowerConfigurationCreatePowerConfiguration

9.4.1 eCLD_PowerConfigurationCreatePowerConfiguration

```
teZCL_Status eCLD_PowerConfigurationCreatePowerConfiguration(
         tsZCL_ClusterInstance *psClusterInstance,
         bool_t bIsServer,
         tsZCL_ClusterDefinition *psClusterDefinition,
         void *pvEndPointSharedStructPtr,
         uint8 *pu8AttributeControlBits);
```

Description

This function creates an instance of the Power Configuration 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 Power Configuration cluster instance on the endpoint, but instances of other clusters may also be created on the same endpoint by calling their corresponding creation functions.

Note: *This function must not be called for an endpoint on which a standard ZigBee device is used. In this case, the device and its supported clusters must be registered on the endpoint using the relevant device registration function.*

When used, this function must be called after the stack has been started and after the ZCL has been initialized.

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 Power Configuration cluster.

The function initializes the array elements to zero.

Parameters

psClusterInstance Pointer to structure containing information about the cluster instance to be created (see Section 6.1.16). This structure is updated by the function by initialising individual structure fields.

bIsServer Type of cluster instance (server or client) to be created:

TRUE - server

FALSE - client

psClusterDefinition Pointer to structure indicating the type of cluster to be created (see Section 6.1.2). In this case, this structure must contain the details of the Power Configuration cluster. This parameter can refer to a pre-filled structure called `sCLD_PowerConfiguration` which is provided in the **PowerConfiguration.h** file.

pvEndPointSharedStructPtr Pointer to the shared structure used for attribute storage. This parameter should be the address of the structure of type `tsCLD_PowerConfiguration` which defines the attributes of Power Configuration cluster. The function initializes the attributes with default values.

pu8AttributeControlBits Pointer to an array of **uint8** values, with one element for each attribute in the cluster (see above).

Returns

E_ZCL_SUCCESS

E_ZCL_ERR_PARAMETER_NULL

9.5 Enumerations and Defines

9.5.1 teCLD_PWRCFG_AttributeId

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

```
typedef enum
{
    /* Mains Information attribute set attribute IDs */
```

```
    E_CLD_PWRCFG_ATTR_ID_MAINS_VOLTAGE                = 0x0000,
    E_CLD_PWRCFG_ATTR_ID_MAINS_FREQUENCY,
    /* Mains settings attribute set attribute IDs */
    E_CLD_PWRCFG_ATTR_ID_MAINS_ALARM_MASK             = 0x0010,
    E_CLD_PWRCFG_ATTR_ID_MAINS_VOLTAGE_MIN_THRESHOLD,
    E_CLD_PWRCFG_ATTR_ID_MAINS_VOLTAGE_MAX_THRESHOLD,
    E_CLD_PWRCFG_ATTR_ID_MAINS_VOLTAGE_DWELL_TRIP_POINT,
    /* Battery information attribute set attribute IDs */
    E_CLD_PWRCFG_ATTR_ID_BATTERY_VOLTAGE              = 0x0020,
    E_CLD_PWRCFG_ATTR_ID_BATTERY_PERCENTAGE_REMAINING,
    /* Battery settings attribute set attribute IDs */
    E_CLD_PWRCFG_ATTR_ID_BATTERY_MANUFACTURER         = 0x0030,
    E_CLD_PWRCFG_ATTR_ID_BATTERY_SIZE,
    E_CLD_PWRCFG_ATTR_ID_BATTERY_AHR_RATING,
    E_CLD_PWRCFG_ATTR_ID_BATTERY_QUANTITY,
    E_CLD_PWRCFG_ATTR_ID_BATTERY_RATED_VOLTAGE,
    E_CLD_PWRCFG_ATTR_ID_BATTERY_ALARM_MASK,
    E_CLD_PWRCFG_ATTR_ID_BATTERY_VOLTAGE_MIN_THRESHOLD,
    E_CLD_PWRCFG_ATTR_ID_BATTERY_VOLTAGE_THRESHOLD1,
    E_CLD_PWRCFG_ATTR_ID_BATTERY_VOLTAGE_THRESHOLD2,
    E_CLD_PWRCFG_ATTR_ID_BATTERY_VOLTAGE_THRESHOLD3,
    E_CLD_PWRCFG_ATTR_ID_BATTERY_PERCENTAGE_MIN_THRESHOLD,
    E_CLD_PWRCFG_ATTR_ID_BATTERY_PERCENTAGE_THRESHOLD1,
    E_CLD_PWRCFG_ATTR_ID_BATTERY_PERCENTAGE_THRESHOLD2,
    E_CLD_PWRCFG_ATTR_ID_BATTERY_PERCENTAGE_THRESHOLD3,
    E_CLD_PWRCFG_ATTR_ID_BATTERY_ALARM_STATE,
    /* Battery information 2 attribute set attribute IDs */
    E_CLD_PWRCFG_ATTR_ID_BATTERY_2_VOLTAGE              = 0x0040,
    E_CLD_PWRCFG_ATTR_ID_BATTERY_2_PERCENTAGE_REMAINING,
    /* Battery settings 2 attribute set attribute IDs */
    E_CLD_PWRCFG_ATTR_ID_BATTERY_2_MANUFACTURER         = 0x0050,
    E_CLD_PWRCFG_ATTR_ID_BATTERY_2_SIZE,
    E_CLD_PWRCFG_ATTR_ID_BATTERY_2_AHR_RATING,
    E_CLD_PWRCFG_ATTR_ID_BATTERY_2_QUANTITY,
    E_CLD_PWRCFG_ATTR_ID_BATTERY_2_RATED_VOLTAGE,
    E_CLD_PWRCFG_ATTR_ID_BATTERY_2_ALARM_MASK,
    E_CLD_PWRCFG_ATTR_ID_BATTERY_2_VOLTAGE_MIN_THRESHOLD,
    E_CLD_PWRCFG_ATTR_ID_BATTERY_2_VOLTAGE_THRESHOLD1,
    E_CLD_PWRCFG_ATTR_ID_BATTERY_2_VOLTAGE_THRESHOLD2,
    E_CLD_PWRCFG_ATTR_ID_BATTERY_2_VOLTAGE_THRESHOLD3,
    E_CLD_PWRCFG_ATTR_ID_BATTERY_2_PERCENTAGE_MIN_THRESHOLD,
    E_CLD_PWRCFG_ATTR_ID_BATTERY_2_PERCENTAGE_THRESHOLD1,
    E_CLD_PWRCFG_ATTR_ID_BATTERY_2_PERCENTAGE_THRESHOLD2,
    E_CLD_PWRCFG_ATTR_ID_BATTERY_2_PERCENTAGE_THRESHOLD3,
    E_CLD_PWRCFG_ATTR_ID_BATTERY_2_ALARM_STATE,
    /* Battery information 3 attribute set attribute IDs */
    E_CLD_PWRCFG_ATTR_ID_BATTERY_3_VOLTAGE              = 0x0060,
    E_CLD_PWRCFG_ATTR_ID_BATTERY_3_PERCENTAGE_REMAINING,
    /* Battery settings 3 attribute set attribute IDs */
    E_CLD_PWRCFG_ATTR_ID_BATTERY_3_MANUFACTURER         = 0x0070,
    E_CLD_PWRCFG_ATTR_ID_BATTERY_3_SIZE,
    E_CLD_PWRCFG_ATTR_ID_BATTERY_3_AHR_RATING,
    E_CLD_PWRCFG_ATTR_ID_BATTERY_3_QUANTITY,
    E_CLD_PWRCFG_ATTR_ID_BATTERY_3_RATED_VOLTAGE,
    E_CLD_PWRCFG_ATTR_ID_BATTERY_3_ALARM_MASK,
    E_CLD_PWRCFG_ATTR_ID_BATTERY_3_VOLTAGE_MIN_THRESHOLD,
    E_CLD_PWRCFG_ATTR_ID_BATTERY_3_VOLTAGE_THRESHOLD1,
    E_CLD_PWRCFG_ATTR_ID_BATTERY_3_VOLTAGE_THRESHOLD2,
    E_CLD_PWRCFG_ATTR_ID_BATTERY_3_VOLTAGE_THRESHOLD3,
    E_CLD_PWRCFG_ATTR_ID_BATTERY_3_PERCENTAGE_MIN_THRESHOLD,
```

```
    E_CLD_PWRCFG_ATTR_ID_BATTERY_3_PERCENTAGE_THRESHOLD1,
    E_CLD_PWRCFG_ATTR_ID_BATTERY_3_PERCENTAGE_THRESHOLD2,
    E_CLD_PWRCFG_ATTR_ID_BATTERY_3_PERCENTAGE_THRESHOLD3,
    E_CLD_PWRCFG_ATTR_ID_BATTERY_3_ALARM_STATE,
    /* Global attribute IDs */
    E_CLD_PWRCFG_ATTR_ID_CLUSTER_REVISION           = 0xFFFC,
    E_CLD_PWRCFG_ATTR_ID_ATTRIBUTE_REPORTING_STATUS = 0xFFFE
} teCLD_PWRCFG_AttributeId;
```

9.5.2 teCLD_PWRCFG_BatterySize

The following structure contains the enumerations used to indicate the type of battery used in the device.

```
typedef enum
{
    E_CLD_PWRCFG_BATTERY_SIZE_NO_BATTERY    = 0x00,
    E_CLD_PWRCFG_BATTERY_SIZE_BUILT_IN,
    E_CLD_PWRCFG_BATTERY_SIZE_OTHER,
    E_CLD_PWRCFG_BATTERY_SIZE_AA,
    E_CLD_PWRCFG_BATTERY_SIZE_AAA,
    E_CLD_PWRCFG_BATTERY_SIZE_C,
    E_CLD_PWRCFG_BATTERY_SIZE_D,
    E_CLD_PWRCFG_BATTERY_SIZE_UNKNOWN       = 0xff,
} teCLD_PWRCFG_BatterySize;
```

9.5.3 Defines for Voltage Alarms

The following #defines are provided for use in the configuration of the mains over-voltage and under-voltage alarms, and the battery-low alarm.

Mains Alarm Mask

```
#define CLD_PWRCFG_MAINS_VOLTAGE_TOO_LOW    (1 << 0)
#define CLD_PWRCFG_MAINS_VOLTAGE_TOO_HIGH   (1 << 1)
```

Battery Alarm Mask

```
#define CLD_PWRCFG_BATTERY_VOLTAGE_TOO_LOW  (1 << 0)
```

9.6 Compile-time options

To enable the Power Configuration cluster in the code to be built, it is necessary to add the following to the **zcl_options.h** file:

```
#define CLD_POWER_CONFIGURATION
```

In addition, to include the software for a cluster client or server or both, it is necessary to add one or both of the following to the same file:

```
#define POWER_CONFIGURATION_CLIENT
#define POWER_CONFIGURATION_SERVER
```

The Power Configuration cluster contains macros that may be optionally specified at compile-time by adding some or all the following lines to the **zcl_options.h** file.

Optional Attributes

Add this line to enable the optional Mains Voltage attribute:

```
#define CLD_PWRCFG_ATTR_MAINS_VOLTAGE
```

Add this line to enable the optional Mains Frequency attribute:

```
#define CLD_PWRCFG_ATTR_MAINS_FREQUENCY
```

Add this line to enable the optional Mains Alarm Mask attribute:

```
#define CLD_PWRCFG_ATTR_MAINS_ALARM_MASK
```

Add this line to enable the optional Mains Voltage Min Threshold attribute:

```
#define CLD_PWRCFG_ATTR_MAINS_VOLTAGE_MIN_THRESHOLD
```

Add this line to enable the optional Mains Voltage Max Threshold attribute:

```
#define CLD_PWRCFG_ATTR_MAINS_VOLTAGE_MAX_THRESHOLD
```

Add this line to enable the optional Mains Voltage Dwell Trip Point attribute:

```
#define CLD_PWRCFG_ATTR_MAINS_VOLTAGE_DWELL_TRIP_POINT
```

Add this line to enable the optional Battery Voltage attribute:

```
#define CLD_PWRCFG_ATTR_BATTERY_VOLTAGE
```

Add this line to enable the optional Battery Manufacturer attributes:

```
#define CLD_PWRCFG_ATTR_BATTERY_MANUFACTURER
```

Add this line to enable the optional Battery Size attribute:

```
#define CLD_PWRCFG_ATTR_BATTERY_SIZE
```

Add this line to enable the optional Battery Amp Hour attribute:

```
#define CLD_PWRCFG_ATTR_BATTERY_AHR_RATING
```

Add this line to enable the optional Battery Quantity attribute:

```
#define CLD_PWRCFG_ATTR_BATTERY_QUANTITY
```

Add this line to enable the optional Battery Rated Voltage attribute:

```
#define CLD_PWRCFG_ATTR_BATTERY_RATED_VOLTAGE
```

Add this line to enable the optional Battery Alarm Mask attribute:

```
#define CLD_PWRCFG_ATTR_BATTERY_ALARM_MASK
```

Add this line to enable the optional Battery Voltage Min Threshold attribute:

```
#define CLD_PWRCFG_ATTR_BATTERY_VOLTAGE_MIN_THRESHOLD
```

Add this line to enable the optional Battery Percentage Life Remaining attribute:

```
#define CLD_PWRCFG_ATTR_BATTERY_PERCENTAGE_REMAINING
```

Add this line to enable the optional Battery Voltage Threshold 1 attribute:

```
#define CLD_PWRCFG_ATTR_ID_BATTERY_VOLTAGE_THRESHOLD1
```

Add this line to enable the optional Battery Voltage Threshold 2 attribute:

```
#define LD_PWRCFG_ATTR_ID_BATTERY_VOLTAGE_THRESHOLD2
```

Add this line to enable the optional Battery Voltage Threshold 3 attribute:

```
#define CLD_PWRCFG_ATTR_ID_BATTERY_VOLTAGE_THRESHOLD3
```

Add this line to enable the optional Battery Percentage Life Min Threshold attribute:

```
#define CLD_PWRCFG_ATTR_ID_BATTERY_PERCENTAGE_MIN_THRESHOLD
```

Add this line to enable the optional Battery Percentage Life Threshold 1 attribute:

```
#define CLD_PWRCFG_ATTR_ID_BATTERY_PERCENTAGE_THRESHOLD1
```

Add this line to enable the optional Battery Percentage Life Threshold 2 attribute:

```
#define CLD_PWRCFG_ATTR_ID_BATTERY_PERCENTAGE_THRESHOLD2
```

Add this line to enable the optional Battery Percentage Life Threshold 3 attribute:

```
#define CLD_PWRCFG_ATTR_ID_BATTERY_PERCENTAGE_THRESHOLD3
```

Add this line to enable the optional Battery Alarm State attribute:

```
#define CLD_PWRCFG_ATTR_ID_BATTERY_ALARM_STATE
```

Add this line to enable the optional Battery <X> Voltage attribute:

```
#define CLD_PWRCFG_ATTR_BATTERY_<X>_VOLTAGE
```

Add this line to enable the optional Battery <X> Percentage Life Remaining attribute:

```
#define CLD_PWRCFG_ATTR_BATTERY_<X>_PERCENTAGE_REMAINING
```

Add this line to enable the optional Battery <X> Manufacturer attributes:

```
#define CLD_PWRCFG_ATTR_BATTERY_<X>_MANUFACTURER
```

Add this line to enable the optional Battery <X> Size attribute:

```
#define CLD_PWRCFG_ATTR_BATTERY_<X>_SIZE
```

Add this line to enable the optional Battery <X> Amp Hour attribute:

```
#define CLD_PWRCFG_ATTR_BATTERY_<X>_AHR_RATING
```

Add this line to enable the optional Battery <X> Quantity attribute:

```
#define CLD_PWRCFG_ATTR_BATTERY_<X>_QUANTITY
```

Add this line to enable the optional Battery <X> Rated Voltage attribute:

```
#define CLD_PWRCFG_ATTR_BATTERY_<X>_RATED_VOLTAGE
```

Add this line to enable the optional Battery <X> Alarm Mask attribute:

```
#define CLD_PWRCFG_ATTR_BATTERY_<X>_ALARM_MASK
```

Add this line to enable the optional Battery <X> Voltage Min Threshold attribute:

```
#define CLD_PWRCFG_ATTR_BATTERY_<X>_VOLTAGE_MIN_THRESHOLD
```

Add this line to enable the optional Battery <X> Voltage Threshold 1 attribute:

```
#define CLD_PWRCFG_ATTR_ID_BATTERY_<X>_VOLTAGE_THRESHOLD1
```

Add this line to enable the optional Battery <X> Voltage Threshold 2 attribute:

```
#define CLD_PWRCFG_ATTR_ID_BATTERY_<X>_VOLTAGE_THRESHOLD2
```

Add this line to enable the optional Battery <X> Voltage Threshold 3 attribute:

```
#define CLD_PWRCFG_ATTR_ID_BATTERY_<X>_VOLTAGE_THRESHOLD3
```

Add this line to enable the optional Battery <X> Percentage Life Remaining attribute:

```
#define CLD_PWRCFG_ATTR_ID_BATTERY_<X>_PERCENTAGE_MIN_THRESHOLD
```

Add this line to enable the optional Battery <X> Percentage Life Threshold 1 attribute:

```
#define CLD_PWRCFG_ATTR_ID_BATTERY_<X>_PERCENTAGE_THRESHOLD1
```

Add this line to enable the optional Battery <X> Percentage Life Threshold 2 attribute:

```
#define CLD_PWRCFG_ATTR_ID_BATTERY_<X>_PERCENTAGE_THRESHOLD2
```

Add this line to enable the optional Battery <X> Percentage Life Threshold 3 attribute:

```
#define CLD_PWRCFG_ATTR_ID_BATTERY_<X>_PERCENTAGE_THRESHOLD3
```

Add this line to enable the optional Battery <X> Alarm State attribute:

```
#define CLD_PWRCFG_ATTR_ID_BATTERY_<X>_ALARM_STATE
```

Add this line to enable the optional Attribute Reporting Status attribute:

```
#define CLD_PWRCFG_ATTR_ID_ATTRIBUTE_REPORTING_STATUS
```

Global Attributes

Add this line to define the value (n) of the Cluster Revision attribute:

```
#define CLD_PWRCFG_CLUSTER_REVISION <n>
```

The default value is 1, which corresponds to the revision of the cluster in the ZCL r6 specification (see Section 2.4).

10 Device Temperature Configuration Cluster

This chapter describes the Device Temperature Configuration cluster, which is concerned with internal temperature of a device.

The Device Temperature Configuration cluster has a Cluster ID of 0x0002.

10.1 Overview

The Device Temperature Configuration cluster allows:

- Information to be obtained about the internal temperature of a device.
- Over-temperature and under-temperature alarms to be configured.

To use the functionality of this cluster, you must include the file **DeviceTemperatureConfiguration.h** in your application and enable the cluster by defining CLD_DEVICE_TEMPERATURE_CONFIGURATION in the **zcl_options.h** file.

It is also necessary to enable the cluster as a server or client, or as both:

- The cluster server is able to receive commands to access internal temperature data on the local device.
- The cluster client is able to send commands to access the internal temperature data on the remote device.

The inclusion of the client or server software must be pre-defined in the compile-time options of the application. In addition, if the cluster is designed to reside on a custom endpoint, then the role of client or server must also be specified when creating the cluster instance.

The compile-time options for the Device Temperature Configuration cluster are fully detailed in Section 10.5.

10.2 Cluster structure and attributes

The structure definition for the Device Temperature Configuration cluster is:

```
typedef struct
{
#ifdef DEVICE_TEMPERATURE_CONFIGURATION_SERVER
    zint16                  i16CurrentTemperature;
#ifdef CLD_DEVTEMPCFG_ATTR_ID_MIN_TEMP_EXPERIENCED
    zint16                  i16MinTempExperienced;
#endif
#ifdef CLD_DEVTEMPCFG_ATTR_ID_MAX_TEMP_EXPERIENCED
    zint16                  i16MaxTempExperienced;
#endif
#ifdef CLD_DEVTEMPCFG_ATTR_ID_OVER_TEMP_TOTAL_DWELL
    zuint16                 u16OverTempTotalDwell;
#endif
#ifdef CLD_DEVTEMPCFG_ATTR_ID_DEVICE_TEMP_ALARM_MASK
    zbmap8                  u8DeviceTempAlarmMask;
#endif
#ifdef CLD_DEVTEMPCFG_ATTR_ID_LOW_TEMP_THRESHOLD
    zint16                  i16LowTempThreshold;
#endif
#ifdef CLD_DEVTEMPCFG_ATTR_ID_HIGH_TEMP_THRESHOLD
    zint16                  i16HighTempThreshold;
#endif
#ifdef CLD_DEVTEMPCFG_ATTR_ID_LOW_TEMP_DWELL_TRIP_POINT
    zuint24                 u24LowTempDwellTripPoint;
#endif
#ifdef CLD_DEVTEMPCFG_ATTR_ID_HIGH_TEMP_DWELL_TRIP_POINT
```

```
    zuint24                 u24HighTempDwellTripPoint;
#endif
#endif
    zuint16                 u16ClusterRevision;
} tsCLD_DeviceTemperatureConfiguration;
```

The attributes are classified into three attribute sets: Device Temperature Information, Device Temperature Settings, and Global. The attributes from these sets are described below.

Device Temperature Information Attribute Set

- `i16CurrentTemperature` is a mandatory attribute representing the current internal temperature of the local device, in degrees Celsius. The valid temperature range is -200 to +200 ^{o}C (and the value is 0xFFFF is invalid).
- `i16MinTempExperienced` is an optional attribute representing the minimum internal temperature experienced by the local device while powered, in degrees Celsius. The valid temperature range is -200 to +200^{o}C (and the value is 0xFFFF is invalid).
- `i16MaxTempExperienced` is an optional attribute representing the maximum internal temperature experienced by the local device while powered, in degrees Celsius. The valid temperature range is -200 to +200^{o}C (and the value is 0xFFFF is invalid).
- `u16OverTempTotalDwell` is an optional attribute representing the total time, in hours, that the device has spent (in its lifetime) above the temperature specified in the attribute `i16HighTempThreshold` (see below).

Device Temperature Settings Attribute Set

- `u8DeviceTempAlarmMask` is an optional attribute containing a bitmap that specifies the device temperature alarms that are enabled and disabled. The relevant bit is set to ‘1’ for alarm enabled and ‘0’ for alarm disabled, as shown in the table below.

Table 29. Device Temperature Settings Attribute bitmap

Bit	Alarm
0	Under-temperature alarm (device temperature too low)
1	Over-temperature alarm (device temperature too high)
2-7	Reserved

- `i16LowTempThreshold` is an optional attribute representing the lower temperature threshold, in degrees Celsius, for an under-temperature alarm. The device temperature is allowed to fall below this threshold for the duration specified by `u24LowTempDwellTripPoint` before the alarm is triggered (see below). 0x8000 indicates that the alarm is not generated.
- `i16HighTempThreshold` is an optional attribute representing the upper temperature threshold, in degrees Celsius, for an over-temperature alarm. The device temperature is allowed to rise above this threshold for the duration specified by `u24HighTempDwellTripPoint` before the alarm is triggered (see below). 0x8000 indicates that the alarm is not generated.
- `u24LowTempDwellTripPoint` is an optional attribute representing the time delay, in seconds, before an under-temperature alarm is triggered when the device temperature falls below the lower threshold value specified in `i16LowTempThreshold`. If the device temperature returns above the threshold during this time, the alarm is canceled. 0xFFFFFF indicates that the under-temperature alarm is not generated.
- `u24HighTempDwellTripPoint` is an optional attribute represent time-delay, in seconds. This attribute specifies the time before an over-temperature alarm is triggered when the device temperature rises above the upper threshold value specified in `i16HighTempThreshold`. If the device temperature returns below the threshold during this time, the alarm is canceled. 0xFFFFFF indicates that the over-temperature alarm is not generated.

Global Attribute Set

`u16ClusterRevision` is a mandatory attribute that specifies the revision of the cluster specification on which this cluster instance is based. The cluster specification in the ZCL r6 corresponds to a cluster revision of 1. The value is incremented by one for each subsequent revision of the cluster specification. This attribute is also described in Section 2.4.

10.3 Functions

The following Device Temperature Configuration cluster function is provided in the NXP implementation of the ZCL:

eCLD_DeviceTemperatureConfigurationCreateDeviceTemperatureConfiguration

10.3.1 eCLD_DeviceTemperatureConfigurationCreateDeviceTemperatureConfiguration

```
teZCL_Status eCLD_DeviceTemperatureConfigurationCreateDeviceTemperatureConfiguration(
        tsZCL_ClusterInstance *psClusterInstance,
        bool_t bIsServer,
        tsZCL_ClusterDefinition *psClusterDefinition,
        void *pvEndPointSharedStructPtr,
        uint8 *pu8AttributeControlBits);
```

Description

This function creates an instance of the Device Temperature Configuration 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 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 creates a Device Temperature Configuration cluster instance on the endpoint, but instances of other clusters may also be created on the same endpoint by calling their corresponding creation functions.

Note: *This function is not called for an endpoint on which a standard ZigBee device is used. In this case, the device and its supported clusters must be registered on the endpoint using the relevant device registration function.*

When used, this function must be called after the stack has been started and after the ZCL has been initialized.

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 Device Temperature Configuration cluster.

The function initializes the array elements to zero.

Parameters

- *psClusterInstance* Pointer to structure containing information about the cluster instance to be created (see Section 6.1.16). This structure is updated by the function by initializing individual structure fields.
- *bIsServer* Type of cluster instance (server or client) to be created: TRUE - server FALSE - client
- *psClusterDefinition* Pointer to structure indicating the type of cluster to be created (see Section 6.1.2). In this case, this structure must contain the details of the Device Temperature Configuration cluster. This parameter can refer to a pre-filled structure called `sCLD_DeviceTemperatureConfiguration` which is provided in the **DeviceTemperatureConfiguration.h** file.

- *pvEndPointSharedStructPtr* Pointer to the shared structure used for attribute storage. This parameter should be the address of the structure of type `tsCLD_DeviceTemperatureConfiguration` which defines the attributes of Device Temperature Configuration cluster. The function initializes the attributes with default values.
- *pu8AttributeControlBits* Pointer to an array of **uint8** values, with one element for each attribute in the cluster (see above).

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL

10.4 Enumerations and Defines

10.4.1 teCLD_DEVTEMPCFG_AttributeId

The following structure contains the enumerations used to identify the attributes of the Device Temperature Configuration cluster.

```
typedef enum
{
    /* Device Temperature Information attribute set attribute IDs  */
    E_CLD_DEVTEMPCFG_ATTR_ID_CURRENT_TEMPERATURE      = 0x0000,
    E_CLD_DEVTEMPCFG_ATTR_ID_MIN_TEMP_EXPERIENCED,
    E_CLD_DEVTEMPCFG_ATTR_ID_MAX_TEMP_EXPERIENCED,
    E_CLD_DEVTEMPCFG_ATTR_ID_OVER_TEMP_TOTAL_DWELL,
    /* Device Temperature Settings attribute set attribute IDs */
    E_CLD_DEVTEMPCFG_ATTR_ID_DEVICE_TEMP_ALARM_MASK   = 0x0010,
    E_CLD_DEVTEMPCFG_ATTR_ID_LOW_TEMP_THRESHOLD,
    E_CLD_DEVTEMPCFG_ATTR_ID_HIGH_TEMP_THRESHOLD,
    E_CLD_DEVTEMPCFG_ATTR_ID_LOW_TEMP_DWELL_TRIP_POINT,
    E_CLD_DEVTEMPCFG_ATTR_ID_HIGH_TEMP_DWELL_TRIP_POINT,
} teCLD_DEVTEMPCFG_AttributeId;
```

10.4.2 Defines for Device Temperature Alarms

The following #defines are provided for use in the configuration of the over-temperature and under-temperature alarms.

```
#define CLD_DEVTEMPCFG_BITMASK_DEVICE_TEMP_TOO_LOW      (1 << 0)
#define CLD_DEVTEMPCFG_BITMASK_DEVICE_TEMP_TOO_HIGH     (1 << 1)
```

10.5 Compile-time options

To enable the Device Temperature Configuration cluster in the code to be built, it is necessary to add the following to the **zcl_options.h** file:

```
#define CLD_DEVICE_TEMPERATURE_CONFIGURATION
```

In addition, to include the software for a cluster client or server or both, it is necessary to add one or both of the following to the same file:

```
#define DEVICE_TEMPERATURE_CONFIGURATION_CLIENT
```

```
#define DEVICE_TEMPERATURE_CONFIGURATION_SERVER
```

The Device Temperature Configuration cluster contains macros that may be optionally specified at compile time by adding some or all the following lines to the **zcl_options.h** file.

Optional Attributes

Add this line to enable the optional Minimum Temperature Experienced attribute:

```
#define CLD_DEVTEMPCFG_ATTR_ID_MIN_TEMP_EXPERIENCED
```

Add this line to enable the optional Maximum Temperature Experienced attribute:

```
#define CLD_DEVTEMPCFG_ATTR_ID_MAX_TEMP_EXPERIENCED
```

Add this line to enable the optional Over Temperature Total attribute:

```
#define CLD_DEVTEMPCFG_ATTR_ID_OVER_TEMP_TOTAL_DWELL
```

Add this line to enable the optional Temperature Alarm Mask attribute:

```
#define CLD_DEVTEMPCFG_ATTR_ID_DEVICE_TEMP_ALARM_MASK
```

Add this line to enable the optional Low Temperature Threshold attribute:

```
#define CLD_DEVTEMPCFG_ATTR_ID_LOW_TEMP_THRESHOLD
```

Add this line to enable the optional High Temperature Threshold attribute:

```
#define CLD_DEVTEMPCFG_ATTR_ID_HIGH_TEMP_THRESHOLD
```

Add this line to enable the optional Low Temperature Trip Point attribute:

```
#define CLD_DEVTEMPCFG_ATTR_ID_LOW_TEMP_DWELL_TRIP_POINT
```

Add this line to enable the optional High Temperature Trip Point attribute:

```
#define CLD_DEVTEMPCFG_ATTR_ID_HIGH_TEMP_DWELL_TRIP_POINT
```

Global Attributes

Add this line to define the value (n) of the Cluster Revision attribute:

```
#define CLD_DEVTEMPCFG_CLUSTER_REVISION <n>
```

The default value is 1, which corresponds to the revision of the cluster in the ZCL r6 specification (see Section 2.4).

11 Identify Cluster

This chapter describes the Identify cluster which allows a device to identify itself (for example, by flashing an LED on the node).

The Identify cluster has a Cluster ID of 0x0003.

11.1 Overview

The Identify cluster allows the host device to be put into identification mode in which the node highlights itself in some way to an observer (in order to distinguish itself from other nodes in the network). It is recommended that identification mode should involve flashing a light with a period of 0.5 seconds.

To use the functionality of this cluster, you must include the file **Identify.h** in your application and enable the cluster by defining CLD_IDENTIFY in the **zcl_options.h** file.

It is also necessary to enable the cluster as a server or client, or as both:

- The cluster server is able to receive commands to start and stop identification mode on the local device.
- The cluster client is able to send the above commands to the server (and therefore control identification mode on the remote device)

The inclusion of the client or server software must be pre-defined in compile-time options of the application. In addition, if the cluster is to reside on a custom endpoint then the role of client or server must also be specified when creating the cluster instance.

The compile-time options for the Identify cluster are fully detailed in Section 11.9.

Note: *The Identify cluster contains optional functionality for the EZ-mode Commissioning module, which is part of the ZigBee Base Device functionality and is described in the ZigBee Devices User Guide (JNUG3131).*

11.2 Identify Cluster Structure and Attribute

The structure definition for the Identify cluster is:

```
typedef struct
{
#ifdef IDENTIFY_SERVER
    zuint16                 u16IdentifyTime;
#ifdef CLD_IDENTIFY_ATTR_COMMISSION_STATE
    zbmap8                  u8CommissionState;
#endif
#endif
    zuint16                 u16ClusterRevision;
} tsCLD_Identify;
```

Where:

- `u16IdentifyTime` is a mandatory attribute specifying the remaining length of time, in seconds, that the device continues in identification mode. Setting the attribute to a non-zero value puts the device into identification mode and the attribute is then decremented every second.
- `u8CommissionState` is an optional attribute for use with EZ-mode Commissioning (see Chapter 40) to indicate the network status and operational status of the node - this information is contained in a bitmap, as follows:

Table 30. u8CommissionState Attribute Bitmap

Bits	Description
0	Network State • 1 if in the correct network (must be 1 if Operational State bit is 1) • 0 if not in a network, or in a temporary network, or network status is unknown
1	Operational State • 1 if commissioned for operation (Network State bit is set to 1) • 0 if not commissioned for operation
2 - 7	Reserved

u16ClusterRevision is a mandatory attribute that specifies the revision of the cluster specification on which this instance is based.
The cluster specification in the ZCL r6 corresponds to a cluster revision of 1. The value is incremented by one for each subs
Section 2.4).

11.3 Initialization

The function **eCLD_IdentifyCreateIdentify()** is used to create an instance of the Identify cluster. This function is called by the initialization function for the host device but can alternatively be used directly by the application in setting up a custom endpoint which supports the Identify cluster (among others).

11.4 Sending Commands

The NXP implementation of the ZCL provides functions for sending commands between an Identify cluster client and server.

11.4.1 Starting and Stopping Identification Mode

The function **eCLD_IdentifyCommandIdentifyRequestSend()** is used on the cluster client to send a command to the cluster server requesting identification mode to be started or stopped on the server device. The required action is contained in the payload of the command (see Section 11.7.2):

- Setting the payload element *u16IdentifyTime* to a non-zero value has the effect of requesting that the server device enters identification mode for a time (in seconds) corresponding to the specified value.
- Setting the payload element *u16IdentifyTime* to zero has the effect of requesting the immediate termination of any identification mode that is in progress on the server device.

Identification mode can alternatively be started and stopped on a light of a remote node as described in Section 11.4.2.

11.4.2 Requesting Identification Effects

The function **eCLD_IdentifyCommandTriggerEffectSend()** can be used to request a particular identification effect or behavior on a light of a remote node. This function can be used for entering and leaving identification mode instead of **eCLD_IdentifyCommandIdentifyRequestSend()**.

The possible behaviors that can be requested are as follows:

- **Blink:** Light is switched on and then off (once)
- **Breathe**: Light is switched on and off by smoothly increasing and then decreasing its brightness over a 1 second period, and then this process is repeated 15 times.
- **Okay:**
  – Colour light goes green for 1 second.

  - Monochrome light flashes twice in 1 second.
- **Channel change:**
  - Colour light goes orange for 8 seconds.
  - Monochrome light switches to maximum brightness for 0.5 s and then to minimum brightness for 7.5 s.
- **Finish effect:** Current stage of effect is completed and then identification mode is terminated (for example, for the Breathe effect, only the current 1 second cycle is completed).
- **Stop effect**: Current effect and identification mode are terminated as soon as possible.

11.4.3 Inquiring about Identification Mode

The function **eCLD_IdentifyCommandIdentifyQueryRequestSend()** is called on an Identify cluster client in order to request a response from a server cluster when it is in identification mode. This request should only be unicast.

11.4.4 Using EZ-mode Commissioning Features

The Identify cluster also contains the following optional features that can be used with EZ-mode commissioning, which is a part of the ZigBee Base Device functionality and is described in the *ZigBee Devices User Guide (JNUG3131)*.

‘EZ-mode Invoke’ Command

The ‘EZ-mode Invoke’ command is supported which allows a device to schedule and start one or more stages of EZ-mode commissioning on a remote device. The command is issued by calling the **eCLD_IdentifyEZModeInvokeCommandSend()** function and allows the following stages to be specified:

1. **Factory Reset:** EZ-mode commissioning configuration of the destination device to be reset to ‘Factory Fresh’ settings.

2. **Network Steering:** Destination device to be put into the ‘Network Steering’ phase.

3. **Find and Bind:** Destination device to be put into the ‘Find and Bind’ phase.

On receiving the command, the event E_CLD_IDENTIFY_CMD_EZ_MODE_INVOKE is generated on the remote device, indicating one or more requested commissioning actions. The local application must perform these actions using the functions of the EZ-mode Commissioning module. If more than one stage is specified, they must be performed sequentially in the above order and must be contiguous.

If the ‘EZ-mode Invoke’ command is to be used by an application, its use must be enabled at compile time (see Section 11.9).

‘Commissioning State’ Attribute

The Identify cluster server contains an optional ‘Commissioning State’ attribute, `u8CommissionState` (see Section 11.2), which indicates whether the local device is:

- a member of the (correct) network
- in a commissioned state and ready for operation

If the ‘Commissioning State’ attribute is to be used by an application, its use must be enabled at compile time (see Section 11.9).

The EZ-mode initiator can send an ‘Update Commission State’ command to the target device in order to update the commissioning state of the target. The command is issued by calling the **eCLD_IdentifyUpdateCommissionStateCommandSend()** function. On receiving this command on the target, the ‘Commissioning

State' attribute is automatically updated. It is good practice for the EZ-mode initiator to send this command to notify the target device when commissioning is complete.

11.5 Sleeping Devices in Identification Mode

In some cases, a device might sleep between activities (for example, a switch that is configured as a sleeping End Device) and is also operating in identification mode. In such a case, the device must wake once per second for the ZCL to decrement the *u16IdentifyTime* attribute (see Section 11.2), which represents the time remaining in identification mode. The device may also use this wake time to highlight itself, for example, flash an LED. The attribute is automatically updated by the ZCL when the application passes an E_ZCL_CBET_TIMER event to the ZCL via the **vZCL_EventHandler()** function. The ZCL also automatically increments ZCL time as a result of this event.

When in identification mode, it is not permissible for a device to sleep for longer than 1 second, and to generate one timer event on waking. Before entering sleep, the value of the *u16IdentifyTime* attribute can be checked. If this value is zero, the device is not in identification mode and is therefore allowed to sleep for longer than 1 second.

For details of updating ZCL time following a prolonged sleep, refer to Section 18.4.1.

11.6 Functions

The following Identify cluster functions are provided in the NXP implementation of the ZCL:

- eCLD_IdentifyCreateIdentify
- eCLD_IdentifyCommandIdentifyRequestSend
- eCLD_IdentifyCommandTriggerEffectSend
- eCLD_IdentifyCommandIdentifyQueryRequestSend
- eCLD_IdentifyEZModeInvokeCommandSend
- eCLD_IdentifyUpdateCommissionStateCommandSend

11.6.1 eCLD_IdentifyCreateIdentify

```
teZCL_Status eCLD_IdentifyCreateIdentify(
        tsZCL_ClusterInstance *psClusterInstance,
        bool_t bIsServer,
        tsZCL_ClusterDefinition *psClusterDefinition,
        void *pvEndPointSharedStructPtr,
        uint8 *pu8AttributeControlBits,
        tsCLD_IdentifyCustomDataStructure
 *psCustomDataStructure);
```

Description

This function creates an instance of the Identify 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 must 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 creates an Identify cluster instance on the endpoint, but instances of other clusters may also be created on the same endpoint by calling their corresponding creation functions.

Note: *This function should not be called for an endpoint on which a standard ZigBee device is used. In this case, the device and its supported clusters must be registered on the endpoint using the relevant device registration function.*

When used, this function must be the first Identify cluster function called in the application, and must be called after the stack has been started and after the ZCL has been initialized.

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 Identify cluster. The function initializes the array elements to zero.

Parameters

- *psClusterInstance*: Pointer to structure containing information about the cluster instance to be created (see Section 6.1.16). This structure is updated by the function by initializing individual structure fields.
- *bIsServer*: Type of cluster instance (server or client) to be created:
  - TRUE - server
  - FALSE - client
- *psClusterDefinition*: Pointer to structure indicating the type of cluster to be created (see Section 6.1.2). In this case, this structure must contain the details of the Identify cluster. This parameter can refer to a pre-filled structure called `sCLD_Identify` which is provided in the **Identify.h** file.
- *pvEndPointSharedStructPtr*: Pointer to the shared structure used for attribute storage. This parameter should be the address of the structure of type `tsCLD_Identify` which defines the attributes of Identify cluster. The function initializes the attributes with default values.
- *pu8AttributeControlBits*: Pointer to an array of **uint8** values, with one element for each attribute in the cluster (see above).
- *psCustomDataStructure*: Pointer to the structure that contains custom data for the Identify cluster (see Section 11.7.1). This structure is used for internal data storage. No knowledge of the fields of this structure is required.

Returns

- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_SUCCESS

11.6.2 eCLD_IdentifyCommandIdentifyRequestSend

```
teZCL_Status eCLD_IdentifyCommandIdentifyRequestSend(
        uint8 u8SourceEndPointId,
        uint8 u8DestinationEndPointId,
        tsZCL_Address *psDestinationAddress,
        uint8 *pu8TransactionSequenceNumber,
        tsCLD_Identify_IdentifyRequestPayload *psPayload);
```

Description

This function can be called on a client device to send a custom command requesting that the recipient server device either enters or exits identification mode. The required action (start or stop identification mode) must be specified in the payload of the custom command (see Section 11.7.2). The required duration of the identification mode is specified in the payload and this value replaces the value in the Identify cluster structure on the target device.

A device which receives this command generates a callback event on the endpoint on which the Identify cluster was registered.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is 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 to send the request. This parameter is used both to send the command and to identify the instance of the shared structure holding the required attribute values.
- *u8DestinationEndPointId*: Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP.
- *psDestinationAddress*: Pointer to a structure holding the address of the node to which the request is sent.
- *pu8TransactionSequenceNumber*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the request.
- *psPayload*: Pointer to a structure containing the payload for the command (see Section 11.7.2).

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

11.6.3 eCLD_IdentifyCommandTriggerEffectSend

```
teZCL_Status eCLD_IdentifyCommandTriggerEffectSend(
        uint8 u8SourceEndPointId,
        uint8 u8DestinationEndPointId,
        tsZCL_Address *psDestinationAddress,
        uint8 *pu8TransactionSequenceNumber,
        teCLD_Identify_EffectId eEffectId,
        uint8 u8EffectVariant);
```

Description

This function can be called on a client device to send a custom command to a server device, in order to control the identification effect on a light of the target node. Therefore, this function can be used to start and stop identification mode instead of **eCLD_IdentifyCommandIdentifyRequestSend()**. Use of the ‘Trigger Effect’ function must be enabled via a compile-time option, as described in Section 11.9.

The following effect commands can be sent using this function:

Effect command	Description
Blink	Light is switched on and then off (once)
Breathe	Light is switched on and off by smoothly increasing and then decreasing its brightness over a 1-second period, and then this is repeated 15 times

Effect command	Description
Okay	• Color light goes green for 1 second • Monochrome light flashes twice in 1 second
Channel change	• Color light goes orange for 8 seconds • Monochrome light switches to maximum brightness for 0.5 s and then to minimum brightness for 7.5 s
Finish effect	Current stage of effect is completed and then identification mode is terminated (for example, for the Breathe effect, only the current 1-second cycle is completed)
Stop effect	Current effect and identification mode are terminated as soon as possible

A variant of the selected effect can also be specified, but currently only the default (as described above) is available.

A device which receives this command generates a callback event on the endpoint on which the Identify cluster was registered.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is 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 to send the request. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values.
- *u8DestinationEndPointId*: Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP.
- *psDestinationAddress*: Pointer to a structure holding the address of the node to which the request is sent.
- *pu8TransactionSequenceNumber*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the request.
- *eEffectId*: Effect command to send (see above), one of:
  - E_CLD_IDENTIFY_EFFECT_BLINK
  - E_CLD_IDENTIFY_EFFECT_BREATHE
  - E_CLD_IDENTIFY_EFFECT_OKAY
  - E_CLD_IDENTIFY_EFFECT_CHANNEL_CHANGE
  - E_CLD_IDENTIFY_EFFECT_FINISH_EFFECT
  - E_CLD_IDENTIFY_EFFECT_STOP_EFFECT
- *u8EffectVariant*: Required variant of specified effect - set to zero for default (as no variants are currently available).

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

11.6.4 eCLD_IdentifyCommandIdentifyQueryRequestSend

```
tsZCL_Status eCLD_IdentifyCommandIdentifyQueryRequestSend(
        uint8 u8SourceEndPointId,
         uint8 u8DestinationEndPointId,
         tsZCL_Address *psDestinationAddress,
         uint8 *pu8TransactionSequenceNumber);
```

Description

This function can be called on a client device to send a custom command requesting a response from any server devices that are currently in identification mode.

A device which receives this command generates a callback event on the endpoint on which the Identify cluster is registered. If the receiving device is in identification mode, it returns a response containing the amount of time for which it continues in this mode (see Section 11.7.3).

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is 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 to send the request. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values.
- *u8DestinationEndPointId*: Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP.
- *psDestinationAddress*: Pointer to a structure holding the address of the node to which the request is sent.
- *pu8TransactionSequenceNumber*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the request.

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

11.6.5 eCLD_IdentifyEZModeInvokeCommandSend

```
teZCL_Status eCLD_IdentifyEZModeInvokeCommandSend(
        uint8 u8SourceEndPointId,
        uint8 u8DestinationEndPointId,
```

```
        tsZCL_Address *psDestinationAddress,
        uint8 *pu8TransactionSequenceNumber,
        bool bDirection,
        tsCLD_Identify_EZModeInvokePayload
*psPayload);
```

Description

This function can be used to send an ‘EZ-mode Invoke’ to a remote device. The sent command requests one or more of the following stages of the EZ-mode commissioning process to be performed on the destination device. EZ-mode commissioning is a part of the ZigBee Base Device functionality and is described in the *ZigBee Devices User Guide (JNUG3131)*).

1. Factory Reset - clears all bindings, group table entries, and the `u8CommissionState` attribute, and reverts to the ‘Factory Fresh’ settings.

2. Network Steering - puts the destination device into the ‘Network Steering’ phase.

3. Find and Bind - puts the destination device into the ‘Find and Bind’ phase.

The required stages are specified in a bitmap in the command payload structure `tsCLD_Identify_EZModeInvokePayload` (see Section 11.7.4). If more than one stage is specified, they must be performed in the above order and be contiguous.

On receiving the ‘EZ-mode Invoke’ command on the destination device, an E_CLD_IDENTIFY_CMD_EZ_MODE_INVOKE event is generated with the required commissioning actions specified in the `u8Action` field of the `tsCLD_Identify_EZModeInvokePayload` structure. It is the responsibility of the local application to perform the requested actions using the functions of the EZ-mode Commissioning module (see Section 40.6).

Note that the ‘EZ-mode Invoke’ command is optional and, if necessary, must be enabled in the compile-time options (see Section 11.9).

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is 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 to send the request. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values.
- *u8DestinationEndPointId*: Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP.
- *psDestinationAddress*: Pointer to a structure holding the address of the node to which the request is sent.
- *pu8TransactionSequenceNumber*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the request.
- *bDirection*: Boolean indicating the direction of the command, as follows (this should always be set to TRUE):
  - TRUE - Identify cluster client to server
  - FALSE - Identify cluster server to client
- *psPayload*: Pointer to a structure containing the payload for the command (see Section 11.7.4).

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL

- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

11.6.6 eCLD_IdentifyUpdateCommissionStateCommandSend

```
teZCL_Status eCLD_IdentifyUpdateCommissionStateCommandSend(
        uint8 u8SourceEndPointId,
        uint8 u8DestinationEndPointId,
        tsZCL_Address *psDestinationAddress,
        uint8 *pu8TransactionSequenceNumber,
        tsCLD_Identify_UpdateCommissionStatePayload
 *psPayload);
```

Description

This function can be used to send an ‘Update Commission State’ command from an EZ-mode initiator device (cluster client) to a target device (cluster server) in order to update the (optional) `u8CommissionState` attribute (see Section 11.2) which is used for EZ-mode commissioning (which is part of the ZigBee Base Device functionality and is described in the *ZigBee Devices User Guide (JNUG3131)*). The command allows individual bits of `u8CommissionState` to be set or cleared (see Section 11.7.4).

On receiving the ‘Update Commission State’ command on the target device, an event is generated and the requested update is automatically performed.

Note that the `u8CommissionState` attribute is optional and, if necessary, must be enabled in the compile-time options (see Section 11.9).

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is 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 to send the request. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values.
- *u8DestinationEndPointId*: Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP.
- *psDestinationAddress*: Pointer to a structure holding the address of the node to which the request is sent.
- *pu8TransactionSequenceNumber*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the request.
- *psPayload*: Pointer to a structure containing the payload for the command (see Section 11.7.4).

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL

- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

11.7 Structures

11.7.1 Custom Data Structure

The Identity cluster requires extra storage space to be allocated to be used by internal functions. The structure definition for this storage is shown below:

```
typedef struct
{
    tsZCL_ReceiveEventAddress                    sReceiveEventAddress;
    tsZCL_CallBackEvent                                sCustomCallBackEvent;
    tsCLD_IdentifyCallBackMessage         sCallBackMessage;
        } tsCLD_IdentifyCustomDataStructure;
```

The fields are for internal use and no knowledge of them is required.

11.7.2 Custom Command Payloads

The following structure contains the payload for an Identify cluster custom command (sent using the function **eCLD_IdentifyCommandIdentifyRequestSend()**):

```
/* Identify request command payload */
typedef struct
{
    zuint16                 u16IdentifyTime;
} tsCLD_Identify_IdentifyRequestPayload;
```

where `u16IdentifyTime` is the amount of time, in seconds, for which the target device is to remain in identification mode. If this element is set to 0x0000 and the target device is in identification mode, the mode is terminated immediately.

11.7.3 Custom Command Responses

The following structure contains the response to a query whether a device is in identification mode (the original query is sent using the function **eCLD_IdentifyCommandIdentifyQueryRequestSend()**):

```
/* Identify query response command payload */
typedef struct
{
    zuint16                 u16Timeout;
} tsCLD_Identify_IdentifyQueryResponsePayload;
```

where `u16Timeout` is the amount of time, in seconds, that the responding device remains in identification mode.

11.7.4 EZ-mode Commissioning Command Payloads

The structures described below may be used when the Identify cluster is used with EZ-mode commissioning (which is part of the ZigBee Base Device functionality and is described in the *ZigBee Devices User Guide (JNUG3131)*).

'EZ-Mode Invoke' Command Payload

The following structure is used when sending an 'EZ-mode Invoke' command (using the **eCLD_Identify EZModeInvokeCommandSend()** function).

```
typedef struct
{
    zbmap8    u8Action;
} tsCLD_Identify_EZModeInvokePayload;
```

where `u8Action` is a bitmap specifying the EZ-mode commissioning actions to be performed on the destination device - a bit is set to '1' if the corresponding action is required, or to '0' if it is not required:

Bits	Action
0	Factory Reset - clears all bindings, group table entries, and the `u8CommissionState` attribute, and reverts to the 'Factory Fresh' settings
1	Network Steering - puts the device into the 'Network Steering' phase
2	Find and Bind - puts the device into the 'Find and Bind' phase
3 - 7	Reserved

'Update Commission State' Command Payload

The following structure is used when sending an 'Update Commission State' command (using the **eCLD_IdentifyUpdateCommissionStateCommandSend()** function), which requests an update to the value of the `u8CommissionState` attribute (for the definition of the attribute, refer to Section 11.2).

```
typedef struct
{
    zenum8    u8Action;
    zbmap8    u8CommissionStateMask;
} tsCLD_Identify_UpdateCommissionStatePayload;
```

where:

- `u8Action` is a value specifying the action to perform (set or clear) on the `u8CommissionState` bits specified through `u8CommissionStateMask`:
  - 1: Set the specified bits to '1'.
  - 2: Clear the specified bits to '0'.

    All other values are reserved.
- `u8CommissionStateMask` is a bitmap in which the bits correspond to the bits of the `u8CommissionState` attribute. A bit of this field indicates whether the corresponding attribute bit is to be updated (according to the action specified in `u8Action`):
  - If a bit is set to '1', the corresponding `u8CommissionState` bit should be updated.
  - If a bit is set to '0', the corresponding `u8CommissionState` bit should not be updated.

11.8 Enumerations

11.8.1 teCLD_Identify_ClusterID

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

```
typedef enum
{
    E_CLD_IDENTIFY_ATTR_ID_IDENTIFY_TIME = 0x0000,    /* Mandatory */
    E_CLD_IDENTIFY_ATTR_ID_COMMISSION_STATE           /* Optional */
 } teCLD_Identify_ClusterID;
```

11.9 Compile-time options

To enable the Identify cluster in the code to be built, it is necessary to add the following to the **zcl_options.h** file:

```
#define CLD_IDENTIFY
```

In addition, to include the software for a cluster client or server or both, it is necessary to add one or both of the following to the same file:

```
#define IDENTIFY_CLIENT
#define IDENTIFY_SERVER
```

The following cluster functionality can be enabled or configured in **zcl_options.h**.

Cluster Revision

Add this line to define the value (n) of the Cluster Revision attribute:

```
#define CLD_IDENTIFY_CLUSTER_REVISION <n>
```

The default value is 1, which corresponds to the revision of the cluster in the ZCL r6 specification (see Section 2.4).

Trigger Effect

Add this line to enable use of the **eCLD_IdentifyCommandTriggerEffectSend()** function to remotely start/stop identification mode:

```
#define CLD_IDENTIFY_CMD_TRIGGER_EFFECT
```

Enhanced Functionality for EZ-mode Commissioning

To enable the optional ‘Commission State’ attribute, you must include:

```
#define CLD_IDENTIFY_ATTR_COMMISSION_STATE
```

To enable the optional ‘EZ-mode Invoke’ command, you must include:

```
#define CLD_IDENTIFY_CMD_EZ_MODE_INVOKE
```

EZ-mode commissioning is part of the ZigBee Base Device functionality and is described in the *ZigBee Devices User Guide (JNUG3131)*.

12 Groups Cluster

This chapter describes the Groups cluster which allows the management of the Group table concerned with group addressing.

The Groups cluster has a Cluster ID of 0x0004.

12.1 Overview

The Groups cluster allows the management of group addressing that is available in ZigBee PRO. In this addressing scheme, an endpoint on a device can be a member of a group comprising endpoints from one or more devices. The group is assigned a 16-bit group ID or address. The group ID and the local member endpoint numbers are held in an entry of the Group table on a device. If a message is sent to a group address, the Group table is used to determine to which endpoints (if any) the message should deliver on the device. A group can be assigned a name of up to 16 characters and the cluster allows the support of group names to be enabled/disabled.

To use the functionality of this cluster, you must include the file **Groups.h** in your application and enable the cluster by defining CLD_GROUPS in the **zcl_options.h** file.

It is also necessary to enable the cluster as a server or client, or as both:

- The cluster server is able to receive commands to modify the local group table.
- The cluster client is able to send commands to the server to request changes to the group table on the server.

The inclusion of the client or server software must be pre-defined in the compile-time options of the application. In addition, if the cluster is to reside on a custom endpoint then the role of client or server must also be specified when creating the cluster instance.

The compile-time options for the Groups cluster are fully detailed in Section 12.8.

12.2 Groups Cluster structure and attributes

The structure definition for the Groups cluster is:

```
typedef struct
{
    zbmap8                  u8NameSupport;
    zuint16                 u16ClusterRevision;
} tsCLD_Groups;
```

where:

- `u8NameSupport` indicates whether group names are supported by the cluster:
  - A most significant bit of 1 indicates that group names are supported.
  - A most significant bit of 0 indicates that group names are not supported.
- `u16ClusterRevision` is a mandatory attribute that specifies the revision of the cluster specification on which this cluster instance is based. The cluster specification in the ZCL r6 corresponds to a cluster revision of 1. The value is incremented by one for each subsequent revision of the cluster specification. This attribute is also described in Section 2.4.

12.3 Initialization

The function **eCLD_GroupsCreateGroups()** is used to create an instance of the Groups cluster. The function is called by the initialization function for the host device.

A local endpoint can be added to a group on the local node using the function **eCLD_GroupsAdd()**. If the group does not exist, the function creates it. Therefore, this is a way of creating a local group.

12.4 Sending Commands

The NXP implementation of the ZCL provides functions for sending commands between a Groups cluster client and server. A command is sent from the client to one or more endpoints on the server. Multiple endpoints can be targeted using binding or group addressing.

12.4.1 Adding Endpoints to Groups

Two functions are provided for adding one or more endpoints to a group on a remote device. Each function sends a command to the endpoints to be added to the group, where the required group is specified in the payload of the command. If the group does not exist in the Group table of the target device, it is added to the table.

- **eCLD_GroupsCommandAddGroupRequestSend()** can be used to request the addition of the target endpoints to the specified group.
- **eCLD_GroupsCommandAddGroupIfIdentifyingRequestSend()** can be used to request the addition of the target endpoints to the specified group if the target device is in identification mode of the Identity cluster (see Chapter 11).

An endpoint can also be added to a local group, as described in Section 12.3.

12.4.2 Removing Endpoints from Groups

Two functions are provided for removing one or more endpoints from groups on a remote device. Each function sends a command to the endpoints to be removed from the groups. If a group is empty following the removal of the endpoint, it is deleted in the Group table.

- **eCLD_GroupsCommandRemoveGroupRequestSend()** can be used to request the removal of the target endpoint from the group which is specified in the payload of the command.
- **eCLD_GroupsCommandRemoveAllGroupsRequestSend()** can be used to request the removal of the target endpoint from all groups on the remote device.

If an endpoint is a member of a scene associated with a group to be removed, the above function calls also results in the removal of the endpoint from the scene.

12.4.3 Obtaining Information about Groups

Two functions are provided for obtaining information about groups. Each function sends a command to the endpoints to which the inquiry relates.

- **eCLD_GroupsCommandViewGroupRequestSend()** can be used to request the name of a group with the ID/address specified in the command payload.
- **eCLD_GroupsCommandGetGroupMembershipRequestSend()** can be used to determine whether the target endpoint is a member of any of the groups specified in the command payload.

12.5 Functions

The following Groups cluster functions are provided in the NXP implementation of the ZCL:

1. eCLD_GroupsCreateGroups
2. eCLD_GroupsAdd
3. eCLD_GroupsCommandAddGroupRequestSend

4. eCLD_GroupsCommandViewGroupRequestSend
5. eCLD_GroupsCommandGetGroupMembershipRequestSend
6. eCLD_GroupsCommandRemoveGroupRequestSend
7. eCLD_GroupsCommandRemoveAllGroupsRequestSend
8. eCLD_GroupsCommandAddGroupIfIdentifyingRequestSend

12.5.1 eCLD_GroupsCreateGroups

```
teZCL_Status eCLD_GroupsCreateGroups(
         tsZCL_ClusterInstance *psClusterInstance,
         bool_t bIsServer,
         tsZCL_ClusterDefinition *psClusterDefinition,
         void *pvEndPointSharedStructPtr,
         tsCLD_GroupsCustomDataStructure
 *psCustomDataStructure,
        tsZCL_EndPointDefinition *psEndPointDefinition);
```

Description

This function creates an instance of the Groups 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 must 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 creates a Groups cluster instance on the endpoint, but instances of other clusters may also be created on the same endpoint by calling their corresponding creation functions.

Note: *Do not call this function for an endpoint on which a standard ZigBee device is used. In this case, the device and its supported clusters must be registered on the endpoint using the relevant device registration function.*

When used, this function must be the first Groups cluster function called in the application, and must be called after the stack has been started and after the ZCL has been initialized.

The function retrieves any group IDs already stored in the Application Information Base (AIB) of the ZigBee PRO stack. However, the AIB does not store group names. If name support is required, the application should store the group names using the NVM module, so that they can be retrieved following a power outage.

Parameters

- *psClusterInstance*: Pointer to structure containing information about the cluster instance to be created (see Section 6.1.16). This structure is updated by the function by initializing individual structure fields.
- *bIsServer*: Type of cluster instance (server or client) to be created:
  - TRUE - server
  - FALSE - client
- *psClusterDefinition*: Pointer to structure indicating the type of cluster to be created (see Section 6.1.2). In this case, this structure must contain the details of the Groups cluster. This parameter can refer to a pre-filled structure called `sCLD_Groups` which is provided in the **Groups.h** file.
- *pvEndPointSharedStructPtr*: Pointer to the shared structure used for attribute storage. This parameter should be the address of the structure of type `tsCLD_Groups` which defines the attributes of Groups cluster. The function initializes the attributes with default values.
- *psCustomDataStructure*: Pointer to a structure containing the storage for internal functions of the cluster (see Section 12.6.1).

- *psEndPointDefinition*: Pointer to the ZCL endpoint definition structure for the application (see Section 6.1.1).

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL

12.5.2 eCLD_GroupsAdd

```
teZCL_Status eCLD_GroupsAdd(uint8 u8SourceEndPointId,
        uint16 u16GroupId,
        uint8 *pu8GroupName);
```

Description

This function adds the specified endpoint on the local node to the group with the specified group ID/address and specified group name. The relevant entry is modified in the Group table on the local endpoint (of the calling application). If the group does not currently exist, it is created by adding a new entry for the group to the Group table.

Note that the number of entries in the Group table must not exceed the value of CLD_GROUPS_MAX_NUMBER_OF_GROUPS defined at compile time (see Section 12.8).

Parameters

- *u8SourceEndPointId*: Number of local endpoint to be added to group.
- *u16GroupId*: 16-bit group ID/address of group.
- *pu8GroupName*: Pointer to character string representing name of group.

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

12.5.3 eCLD_GroupsCommandAddGroupRequestSend

```
teZCL_Status eCLD_GroupsCommandAddGroupRequestSend(
         uint8 u8SourceEndPointId,
         uint8 u8DestinationEndPointId,
         tsZCL_Address *psDestinationAddress,
         uint8 *pu8TransactionSequenceNumber,
         tsCLD_Groups_AddGroupRequestPayload
 *psPayload);
```

Description

This function sends an Add Group command to a remote device, requesting that the specified endpoints on the target device be added to a group. The group ID/address and name (if supported) are specified in the payload of the message, and must be added to the Group table on the target node along with the associated endpoint numbers.

The device receiving this message generates a callback event on the endpoint on which the Groups cluster is registered. Also, add the group to its Group table before sending a response indicating success or failure (see Section 12.6.4).

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is 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 to send the request. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values.
- *u8DestinationEndPointId*: Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP.
- *psDestinationAddress*: Pointer to a structure holding the address of the node to which the request is sent.
- *pu8TransactionSequenceNumber*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the request.
- *psPayload*: Pointer to a structure containing the payload for this message (see Section 12.6.3).

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

12.5.4 eCLD_GroupsCommandViewGroupRequestSend

```
teZCL_Status eCLD_GroupsCommandViewGroupRequestSend(
         uint8 u8SourceEndPointId,
         uint8 u8DestinationEndPointId,
         tsZCL_Address *psDestinationAddress,
         uint8 *pu8TransactionSequenceNumber,
         tsCLD_Groups_ViewGroupRequestPayload
 *psPayload);
```

Description

This function sends a View Group command to a remote device, requesting the name of the group with the specified group ID (address) on the destination endpoint.

The device receiving this message generates a callback event on the endpoint on which the Groups cluster was registered and generates a View Group response containing the group name (see Section 12.6.4).

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is 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 to send the request. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values.
- *u8DestinationEndPointId*: Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP.
- *psDestinationAddress*: Pointer to a structure holding the address of the node to which the request is sent.
- *pu8TransactionSequenceNumber*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the request.
- *psPayload*: Pointer to a structure containing the payload for this message (see Section 12.6.3).

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

12.5.5 eCLD_GroupsCommandGetGroupMembershipRequestSend

```
teZCL_Status eCLD_GroupsCommandGetGroupMembershipRequestSend
        (uint8 u8SourceEndPointId,
         uint8 u8DestinationEndPointId,
         tsZCL_Address *psDestinationAddress,
         uint8 *pu8TransactionSequenceNumber,
        tsCLD_Groups_GetGroupMembershipRequestPayload
 *psPayload);
```

Description

This function sends a Get Group Membership command to inquire whether the target endpoint is a member of any of the groups specified in a list contained in the command payload.

The device receiving this message generates a callback event on the endpoint on which the Groups cluster is registered and generates a Get Group Membership response containing the required information (see Section 12.6.4).

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is 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 to send the request. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values.
- *u8DestinationEndPointId*: Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP.
- *psDestinationAddress*: Pointer to a structure holding the address of the node to which the request is sent.
- *pu8TransactionSequenceNumber*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the request.
- *psPayload*: Pointer to a structure containing the payload for this message (see Section 12.6.3).

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

12.5.6 eCLD_GroupsCommandRemoveGroupRequestSend

```
teZCL_Status eCLD_GroupsCommandRemoveGroupRequestSend(
          uint8 u8SourceEndPointId,
          uint8 u8DestinationEndPointId,
          tsZCL_Address *psDestinationAddress,
          uint8 *pu8TransactionSequenceNumber,
         tsCLD_Groups_RemoveGroupRequestPayload
 *psPayload);
```

Description

This function sends a Remove Group command to request that the target device deletes membership of the destination endpoints from a particular group - that is, remove the endpoints from the entry of the group in the Group table on the device and, if no other endpoints remain in the group, remove the group from the table.

The device receiving this message generates a callback event on the endpoint on which the Groups cluster is registered. If the group becomes empty following the deletion, the device removes the group ID and group name from its Group table. It then generates an appropriate Remove Group response indicating success or failure (see Section 12.6.4).

If the target endpoint belongs to a scene associated with the group to be removed (requiring the Scenes cluster - see Chapter 13), the endpoint is also removed from this scene as a result of this function call - that is, the relevant scene entry is deleted from the Scene table on the target device.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is 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 to send the request. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values.
- *u8DestinationEndPointId*: The number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP.
- *psDestinationAddress*: Pointer to a structure holding the address of the node to which the request is sent.
- *pu8TransactionSequenceNumber*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the request.
- *psPayload*: Pointer to a structure containing the payload for this message (see Section 12.6.3).

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

12.5.7 eCLD_GroupsCommandRemoveAllGroupsRequestSend

```
teZCL_Status eCLD_GroupsCommandRemoveAllGroupsRequestSend
         uint8 u8SourceEndPointId,
         uint8 u8DestinationEndPointId,
         tsZCL_Address *psDestinationAddress,
         uint8 *pu8TransactionSequenceNumber);
```

Description

This function sends a **Remove All Groups** command to request that the target device removes all group memberships of the destination endpoints. Issuing this command removes the endpoints from all group entries in the Group table on the device. If no other endpoints remain in a group, the function removes the group from the table.

The device receiving this message generates a callback event on the endpoint on which the Groups cluster is registered. If a group becomes empty following the deletion, the device removes the group ID and group name from its Group table.

If the target endpoint belongs to scenes associated with the groups to be removed, calling this function also removes the endpoint from the scenes. The relevant scene entries are deleted from the Scene table on the target device. (For details about the Scenes cluster - refer to Chapter 13).

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is 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 to send the request. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values.
- *u8DestinationEndPointId*: The number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP.
- *psDestinationAddress*: Pointer to a structure holding the address of the node to which the request is sent.
- *pu8TransactionSequenceNumber*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the request.

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

The `eCLD_GroupsCommandRemoveAllGroupsRequestSend` function invokes the ZigBee PRO stack function to transmit the data. If the ZigBee PRO stack function returns an error, the same can be obtained by calling the **eZCL_GetLastZpsError()** function.

12.5.8 eCLD_GroupsCommandAddGroupIfIdentifyingRequestSend

```
teZCL_Status eCLD_GroupsCommandAddGroupIfIdentifyingRequestSend
        (uint8 u8SourceEndPointId,
         uint8 u8DestinationEndPointId,
         tsZCL_Address *psDestinationAddress,
         uint8 *pu8TransactionSequenceNumber,
         tsCLD_Groups_AddGroupRequestPayload
 *psPayload);
```

Description

This function sends an Add Group If Identifying command to a remote device, requesting that the specified endpoints on the target device be added to a particular group on the condition that the remote device is identifying itself. The group ID/address and name (if supported) are specified in the payload of the message, and must be added to the Group table on the target node along with the associated endpoint numbers. The identifying functionality is controlled using the Identify cluster (see Chapter 11).

The device receiving this message generates a callback event on the endpoint on which the Groups cluster is registered and then checks whether the device is identifying itself. If so, the device (if possible) adds the group ID and group name to its Group table. If the device is not identifying itself, then no action is taken.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is 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 to send the request. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values.
- *u8DestinationEndPointId*: Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP.
- *psDestinationAddress*: Pointer to a structure holding the address of the node to which the request is sent.
- *pu8TransactionSequenceNumber*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the request.
- *psPayload*: Pointer to a structure containing the payload for this message (see Section 12.6.3).

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function, which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

12.6 Structures

12.6.1 Custom Data Structure

The Groups cluster requires extra storage space to be allocated to be used by internal functions. The structure definition for this storage is shown below:

```
typedef struct
{
    DLIST                    lGroupsAllocList;
    DLIST                    lGroupsDeAllocList;
    bool                    bIdentifying;
    tsZCL_ReceiveEventAddress    sReceiveEventAddress;
    tsZCL_CallBackEvent          sCustomCallBackEvent;
    tsCLD_GroupsCallBackMessage  sCallBackMessage;
#if (defined CLD_GROUPS) && (defined GROUPS_SERVER)
    tsCLD_GroupTableEntry
          asGroupTableEntry[CLD_GROUPS_MAX_NUMBER_OF_GROUPS];
#endif
} tsCLD_GroupsCustomDataStructure;
```

The fields are for internal use and no knowledge of them is required.

However, the structure `tsCLD_GroupTableEntry` used for the Group table entries is shown in Section 12.6.2.

12.6.2 Group Table Entry

The following structure contains a Group table entry.

```
typedef struct
{
    DNODE   dllGroupNode;
    uint16  u16GroupId;
    uint8   au8GroupName[CLD_GROUPS_MAX_GROUP_NAME_LENGTH + 1];
} tsCLD_GroupTableEntry;
```

The fields are for internal use and no knowledge of them is required.

12.6.3 Custom Command Payloads

The following structures contain the payloads for the Groups cluster custom commands.

Add Group Request Payload

```
typedef struct
{
    zuint16                 u16GroupId;
    tsZCL_CharacterString   sGroupName;
} tsCLD_Groups_AddGroupRequestPayload;
```

where:

- `u16GroupId` is the ID/address of the group to which the endpoints must be added.
- `sGroupName` is the name of the group to which the endpoints must be added.

View Group Request Payload

```
typedef struct
{
    zuint16             u16GroupId;
} tsCLD_Groups_ViewGroupRequestPayload;
```

where `u16GroupId` is the ID/address of the group whose name is required

Get Group Membership Request Payload

```
typedef struct
{
    zuint8              u8GroupCount;
    zint16              *pi16GroupList;
} tsCLD_Groups_GetGroupMembershipRequestPayload;
```

where:

- `u8GroupCount` is the number of groups in the list of the next field.

- `pi16GroupList` is a pointer to a list of groups whose memberships are being queried, where each group is represented by its group ID/address.

Remove Group Request Payload

```
typedef struct
{
    zuint16             u16GroupId;
} tsCLD_Groups_RemoveGroupRequestPayload;
```

where `u16GroupId` is the ID/address of the group from which the endpoints must be removed.

12.6.4 Custom Command Responses

The Groups cluster generates responses to certain custom commands. The responses which contain payloads are detailed below:

Add Group Response Payload

```
typedef struct
{
    zenum8              eStatus;
    zuint16              u16GroupId;
} tsCLD_Groups_AddGroupResponsePayload;
```

where:

- `eStatus` is the status (success or failure) of the requested group addition.
- `u16GroupId` is the ID/address of the group to which endpoints were added.

View Group Response Payload

```
typedef struct
{
    zenum8                  eStatus;
    zuint16                 u16GroupId;
    tsZCL_CharacterString   sGroupName;
} tsCLD_Groups_ViewGroupResponsePayload;
```

where:

- `eStatus` is the status (success or failure) of the requested operation.
- `u16GroupId` is the ID/address of the group whose name was requested.
- `sGroupName` is the returned name of the specified group.

Get Group Membership Response Payload

```
typedef struct
{
    zuint8                  u8Capacity;
    zuint8                  u8GroupCount;
    zint16                 *pi16GroupList;
} tsCLD_Groups_GetGroupMembershipResponsePayload;
```

where:

- `u8Capacity` is the capacity of the Group table of the device to receive more groups - that is, the number of groups that may be added (special values: 0xFE means that at least one additional group may be added, a higher value means that the remaining capacity of the table is unknown).
- `u8GroupCount` is the number of groups in the list of the next field.
- `pi16GroupList` is a pointer to the returned list of groups from those queried that exist on the device, where each group is represented by its group ID/address.

Remove Group Response Payload

```
typedef struct
{
    zenum8                eStatus;
    zuint16               u16GroupId;
} tsCLD_Groups_RemoveGroupResponsePayload;
```

where:

- `eStatus` is the status (success or failure) of the requested group modification.
- `u16GroupId` is the ID/address of the group from which endpoints were removed.

12.7 Enumerations

12.7.1 teCLD_Groups_ClusterID

The following structure contains the enumeration used to identify the attribute of the Groups cluster.

```
typedef enum
{
    E_CLD_GROUPS_ATTR_ID_NAME_SUPPORT        = 0x0000   /* Mandatory */
} teCLD_Groups_ClusterID;
```

12.8 Compile-time Options

To enable the Groups cluster in the code to be built, it is necessary to add the following to the **zcl_options.h** file:

```
#define CLD_GROUPS
```

In addition, to include the software for a cluster client or server or both, it is necessary to add one or both of the following to the same file:

```
#define GROUPS_CLIENT
#define GROUPS_SERVER
```

The Groups cluster contains macros that may be optionally specified at compile time by adding one or both of the following lines to the **zcl_options.h** file.

To set the size used for the group addressing table in the **.zpscfg** file,

Add this line:

```
#define CLD_GROUPS_MAX_NUMBER_OF_GROUPS             (8)
```

To configure the maximum length of the group name, add the following line:

```
#define CLD_GROUPS_MAX_GROUP_NAME_LENGTH            (16)
```

To define the value (n) of the Cluster Revision attribute, add the following line:

```
#define CLD_GROUPS_CLUSTER_REVISION <n>
```

The default value is 1, which corresponds to the revision of the cluster in the ZCL r6 specification (see Section 2.4).

13 Scenes Cluster

This chapter describes the Scenes cluster that allows scenes to be managed.

The Scenes cluster has a Cluster ID of 0x0005.

13.1 Overview

A scene is a set of stored attribute values for one or more cluster instances, where these cluster instances may exist on endpoints on one or more devices.

The Scenes cluster allows standard values for these attributes to be set and retrieved. Thus, the cluster can be used to put the network or part of the network into a pre-defined mode (for example, Night or Day mode for a lighting network). These pre-defined scenes can be used as a basis for 'mood lighting'. A Scenes cluster instance must be created on each endpoint which contains a cluster that is part of a scene.

A scene is often associated with a group (which collects together a set of endpoints over one or more devices) - groups are described in Chapter 12. A scene may, however, be used without a group.

Note: *When the Scenes cluster is used on an endpoint, a Groups cluster instance must always be created on the same endpoint, even if a group is not used for the scene.*

If a cluster on a device is used in a scene, an entry for the scene must be contained in the Scene table on the device. A Scene table entry includes the scene ID, the group ID associated with the scene (0x0000 if there is no associated group), the scene transition time (amount of time to switch to the scene), and the attribute settings for the clusters on the device. The scene ID must be unique within the group with which the scene is associated.

To use the functionality of this cluster, you must include the file **Scenes.h** in your application and enable the cluster by defining CLD_SCENES in the **zcl_options.h** file.

It is also necessary to enable the cluster as a server or client, or as both:

- The cluster server is able to receive commands to access scenes.
- The cluster client is able to send commands to the server to request read or write access to scenes.

The inclusion of the client or server software must be pre-defined in the compile-time options of the application (in addition, if the cluster is to reside on a custom endpoint then the role of client or server must also be specified when creating the cluster instance).

The compile-time options for the Scenes cluster are fully detailed in Section 13.9.

13.2 Scenes Cluster structure and attributes

The structure definition for the Scenes cluster is:

```
typedef struct
{
#ifdef SCENES_SERVER
   zuint8  u8SceneCount;
   zuint8  u8CurrentScene;
   zuint16 u16CurrentGroup;
   zbool   bSceneValid;
   zuint8  u8NameSupport;
#ifdef CLD_SCENES_ATTR_LAST_CONFIGURED_BY
   zieeeaddress  u64LastConfiguredBy;
#endif
#endif
    zuint16 u16ClusterRevision;
```

```
} tsCLD_Scenes;
```

where:

- `u8SceneCount` is the number of scenes currently in the Scene table.
- `u8CurrentScene` is the scene ID of the last scene invoked on the device.
- `u16CurrentGroup` is the group ID of the group associated with the last scene invoked (or 0x0000 if this scene is not associated with a group).
- `bSceneValid` indicates whether the current state of the device corresponds to the values of the `CurrentScene` and `CurrentGroup` attributes (TRUE if they do, FALSE if they do not).
- `u8NameSupport` indicates whether scene names are supported - if the most significant bit is 1 then they are supported, otherwise they are not supported.
- `u64LastConfiguredBy` is the 64-bit IEEE address of the device that last configured the Scene table (0xFFFFFFFFFFFFFFFF indicates that the address is unknown or the table has not been configured).
- `u16ClusterRevision` is a mandatory attribute that specifies the revision of the cluster specification on which this cluster instance is based. The cluster specification in the ZCL r6 corresponds to a cluster revision of 1. The value is incremented by one for each subsequent revision of the cluster specification. This attribute is also described in Section 2.4.

13.3 Initialization

The function **eCLD_ScenesCreateScenes()** is used to create an instance of the Scenes cluster. The function is generally called by the initialization function for the host device.

13.4 Sending Remote Commands

The NXP implementation of the ZCL provides functions for sending commands between a Scenes cluster client and server. A command is sent from the client to one or more endpoints on the server. Multiple endpoints can usually be targeted using binding or group addressing.

Note: *Commands can also be issued for operations on the local node, as described in Section 13.5.*

13.4.1 Creating a Scene

In order to create a scene, add an entry for the scene to the Scene table on every device that contains a cluster, which is associated with the scene.

Use the function **eCLD_ScenesCommandAddSceneRequestSend()** to request a scene to be added to a Scene table on a remote device. Invoking this function sends a request to a single device or to multiple devices (using binding or group addressing). The fields of the Scene table entry are specified in the payload of the request.

Alternatively:

- The function **eCLD_ScenesCommandEnhancedAddSceneRequestSend()** can be used to request that a scene is added to a Scene table on a remote device. This method allows the transition time for the scene to be set in units of tenths of a second (rather than seconds).
- A scene can be created by saving the current attribute settings of the relevant clusters. In this way, the current state of the system can be captured as a scene and reapplied ‘at the touch of a button’ when required. For example scenes can be created for lighting levels in a ‘smart lighting’ system. The current settings are stored as a scene in the Scene table using **eCLD_ScenesCommandStoreSceneRequestSend()**. This function can send the request to a single device or multiple devices. If a Scene table entry exists with the same scene ID and group ID, the existing cluster settings in the entry are overwritten with the new ‘captured’ settings.

Note: *This operation of capturing the current system state as a scene does not result in meaningful settings for the transition time and scene name fields of the Scene table entry. If non-null values are required for these fields, the table entry should be created in advance with the desired field values using* ***eCLD_ScenesCommandAddSceneRequestSend()****.*

13.4.2 Copying a Scene

Scene settings can be copied from one scene to another scene on the same remote endpoint using the **eCLD_ScenesCommandCopySceneSceneRequestSend()** function. This function allows the settings from an existing scene with a specified source scene ID and associated group ID to be copied to a new scene with a specified destination scene ID and associated group ID.

Note: *If an entry corresponding to the target scene ID and group ID exists in the Scene table on the endpoint, the entry settings are overwritten with the copied settings. Otherwise, a new Scene table entry is created with these settings.*

The above function also allows all scenes associated with particular group ID to be copied to another group ID. In this case, the original scene IDs are maintained but are associated with the new group ID (any specified source and destination scene IDs are ignored). Thus, the same scene IDs are associated with two different group IDs.

13.4.3 Applying a Scene

The cluster settings of a scene stored in the Scene table can be retrieved and applied to the system by calling **eCLD_ScenesCommandRecallSceneRequestSend()**. Again, this function can send a request to a single device or to multiple devices (using binding or group addressing).

If the required scene does not contain any settings for a particular cluster or there are some missing attribute values for a cluster, these attribute values remain unchanged in the implementation of the cluster - that is, the corresponding parts of the system do not change their states.

13.4.4 Deleting a Scene

Two functions are provided for removing scenes from the system:

- **eCLD_ScenesCommandRemoveSceneRequestSend()** can be used to request the removal of the destination endpoint from a particular scene - that is, to remove the scene from the Scene table on the target device.
- **eCLD_ScenesCommandRemoveAllScenesRequestSend()** can be used to request that the target device removes scenes associated with a particular group ID/address - that is, remove all Scene table entries relating to this group ID. Specifying a group ID of 0x0000 removes all scenes not associated with a group.

13.4.5 Obtaining Information about Scenes

The following functions are provided for obtaining information about scenes:

- **eCLD_ScenesCommandViewSceneRequestSend()** can be used to request information on a particular scene on the destination endpoint. Only one device may be targeted by this function. The target device returns a response containing the relevant information.

  Alternatively, **eCLD_ScenesCommandEnhancedViewSceneRequestSend()** can be used, which allows the transition time for the scene to be obtained in units of tenths of a second (rather than seconds).
- **eCLD_ScenesCommandGetSceneMembershipRequestSend()** can be used to discover which scenes are associated with a particular group on a device. The request can be sent to a single device or to multiple devices. The target device returns a response containing the relevant information (in the case of multiple

target devices, no response is returned from a device that does not contain a scene associated with the specified group ID). In this way, the function can be used to determine the unused scene IDs.

13.5 Issuing Local Commands

Some of the operations described in Section 13.4 that correspond to remote commands can also be performed locally, as described below.

13.5.1 Creating a Scene

A scene can be created on the local node using either of the following functions:

- **eCLD_ScenesAdd():** This function can be used to add a new scene to the Scene table on the specified local endpoint. A scene ID and an associated group ID must be specified (the latter must be set to 0x0000 if there is no group association). If a scene with these IDs exists in the table, the existing entry is overwritten.
- **eCLD_ScenesStore():** This function can be used to save the currently implemented attribute values on the device to a scene in the Scene table on the specified local endpoint. A scene ID and an associated group ID must be specified (the latter must be set to 0x0000 if there is no group association). If a scene with these IDs exists in the table, the existing entry is overwritten except for the transition time and scene name fields.

13.5.2 Applying a Scene

An existing scene can be applied on the local node using the function **eCLD_ScenesRecall()**. This function reads the stored attribute values for the specified scene from the local Scene table and implements them on the device. The values of any attributes that are not included in the scene remain unchanged.

13.6 Functions

The following Scenes cluster functions are provided in the NXP implementation of the ZCL:

1. eCLD_ScenesCreateScenes
2. eCLD_ScenesAdd
3. eCLD_ScenesStore
4. eCLD_ScenesRecall
5. eCLD_ScenesCommandAddSceneRequestSend
6. eCLD_ScenesCommandViewSceneRequestSend
7. eCLD_ScenesCommandRemoveSceneRequestSend
8. eCLD_ScenesCommandRemoveAllScenesRequestSend
9. eCLD_ScenesCommandStoreSceneRequestSend
10. eCLD_ScenesCommandRecallSceneRequestSend
11. eCLD_ScenesCommandGetSceneMembershipRequestSend
12. eCLD_ScenesCommandEnhancedAddSceneRequestSend
13. eCLD_ScenesCommandEnhancedViewSceneRequestSend
14. eCLD_ScenesCommandCopySceneSceneRequestSend

13.6.1 eCLD_ScenesCreateScenes

```
teZCL_Status eCLD_ScenesCreateScenes(
         tsZCL_ClusterInstance *psClusterInstance,
         bool_t bIsServer,
         tsZCL_ClusterDefinition *psClusterDefinition,
         void *pvEndPointSharedStructPtr,
```

```
          uint8 *pu8AttributeControlBits,
          tsCLD_ScenesCustomDataStructure
   *psCustomDataStructure,
         tsZCL_EndPointDefinition *psEndPointDefinition);
```

Description

This function creates an instance of the Scenes 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 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 creates a Scenes cluster instance on the endpoint, but instances of other clusters may also be created on the same endpoint by calling their corresponding creation functions.

Note: *Do not call this function for an endpoint on which a standard ZigBee device is used. In this case, the device and its supported clusters must be registered on the endpoint using the relevant device registration function.*

When used, this function must be the first Scenes cluster function called in the application, and must be called after the stack has been started and after the ZCL has been initialized.

On calling this function for the first time, a ‘global scene’ entry is created/reserved in the Scene table. On subsequent calls (for example, following a power-cycle or on waking from sleep), if the scene data is recovered by the application from non-volatile memory before the function is called then there is no reinitialization of the scene data. Note that removing all groups from the device also removes the global scene entry (along with other scene entries) from the Scene table.

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 Scene cluster.

The function initializes the array elements to zero.

Parameters

- *psClusterInstance*: Pointer to structure containing information about the cluster instance to be created (see Section 6.1.16). This structure is updated by the function by initializing individual structure fields.
- *bIsServer*: Type of cluster instance (server or client) to be created:
  - TRUE - server
  - FALSE - client
- *psClusterDefinition*: Pointer to structure indicating the type of cluster to be created (see Section 6.1.2). In this case, this structure must contain the details of the Scenes cluster. This parameter can refer to a pre-filled structure called `sCLD_Scenes` which is provided in the **Scenes.h** file.
- *pvEndPointSharedStructPtr*: Pointer to the shared structure used for attribute storage. This parameter should be the address of the structure of type `tsCLD_Scenes` which defines the attributes of Scenes cluster. The function initializes the attributes with default values.
- *pu8AttributeControlBits*: Pointer to an array of **uint8** values, with one element for each attribute in the cluster (see above).
- *psCustomDataStructure*: Pointer to a structure containing the storage for internal functions of the cluster (see Section 13.7.1)
- *psEndPointDefinition*: Pointer to the ZCL endpoint definition structure for the application (see Section 6.1.1)

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL

13.6.2 eCLD_ScenesAdd

```
teZCL_Status eCLD_ScenesAdd(
        uint8 u8SourceEndPointId,
        uint16 u16GroupId,
        uint8 u8SceneId);
```

Description

This function adds a new scene on the specified local endpoint - that is, adds an entry to the Scenes table on the endpoint. The group ID associated with the scene must also be specified (or set to 0x0000 if there is no associated group).

If a scene with the specified scene ID and group ID exists in the table, the existing entry is overwritten (that is, all previous scene data in this entry is lost).

Parameters

- *u8SourceEndPointId*: Number of local endpoint on which Scene table entry is to be added.
- *u16GroupId*: 16-bit group ID/address of associated group (or 0x0000 if no group).
- *u8SceneId*: 8-bit scene ID of new scene.

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL

13.6.3 eCLD_ScenesStore

```
teZCL_Status eCLD_ScenesStore(
        uint8 u8SourceEndPointId,
        uint16 u16GroupId,
        uint8 u8SceneId);
```

Description

This function adds a new scene on the specified local endpoint, based on the current cluster attribute values of the device- that is, saves the current attribute values of the device to a new entry of the Scenes table on the endpoint. The group ID associated with the scene must also be specified (or set to 0x0000 if there is no associated group).

If a scene with the specified scene ID and group ID exists in the table, the existing entry is overwritten. The previous scene data in this entry is lost, except for the transition time field and the scene name field - these fields are left unchanged.

Parameters

- *u8SourceEndPointId*: Number of local endpoint on which Scene table entry is to be added.
- *u16GroupId:* 16-bit group ID/address of associated group (or 0x0000 if no group)
- *u8SceneId*: 8-bit scene ID of scene

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL

13.6.4 eCLD_ScenesRecall

```
teZCL_Status eCLD_ScenesRecall(
        uint8 u8SourceEndPointId,
        uint16 u16GroupId,
        uint8 u8SceneId);
```

Description

This function obtains the attribute values (from the extension fields) of the scene with the specified Scene ID and Group ID on the specified (local) endpoint, and sets the corresponding cluster attributes on the device to these values. Thus, the function reads the stored attribute values for a scene and implements them on the device.

Note that the values of any cluster attributes that are not included in the scene remains unchanged.

Parameters

- *u8SourceEndPointId*: Number of local endpoint containing Scene table to be read.
- *u16GroupId*: 16-bit group ID/address of associated group (or 0x0000 if no group)
- *u8SceneId*: 8-bit scene ID of scene to be read.

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL

13.6.5 eCLD_ScenesCommandAddSceneRequestSend

```
teZCL_Status eCLD_ScenesCommandAddSceneRequestSend(
         uint8 u8SourceEndPointId,
         uint8 u8DestinationEndPointId,
         tsZCL_Address *psDestinationAddress,
         uint8 *pu8TransactionSequenceNumber,
         tsCLD_ScenesAddSceneRequestPayload *psPayload);
```

Description

This function sends an Add Scene command to a remote device in order to add a scene on the specified endpoint - that is, to add an entry to the Scene table on the endpoint. The scene ID is specified in the payload of

the message, along with a duration for the scene among other values (see Section 13.7.2). The scene may also be associated with a particular group.

The device receiving this message generates a callback event on the endpoint on which the Scenes cluster is registered and, if possible, add the scene to its Scene table before sending an Add Scene response indicating success or failure (see Section 13.7.3).

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is 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 to send the request. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values.
- *u8DestinationEndPointId*: Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP.
- *psDestinationAddress*: Pointer to a structure holding the address of the node to which the request is sent.
- *pu8TransactionSequenceNumber*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the request.
- *psPayload*: Pointer to a structure containing the payload for this message (see Section 13.7.2)

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

13.6.6 eCLD_ScenesCommandViewSceneRequestSend

```
teZCL_Status eCLD_ScenesCommandViewSceneRequestSend(
         uint8 u8SourceEndPointId,
         uint8 u8DestinationEndPointId,
         tsZCL_Address *psDestinationAddress,
         uint8 *pu8TransactionSequenceNumber,
         tsCLD_ScenesViewSceneRequestPayload
 *psPayload);
```

Description

This function sends a View Scene command to a remote device, requesting information on a particular scene on the destination endpoint. The relevant scene ID is specified in the command payload. Note that this command can only be sent to an individual device/endpoint and not to a group address.

The device receiving this message generates a callback event on the endpoint on which the Scenes cluster was registered and generates a View Scene response containing the relevant information (see Section 13.7.3).

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is 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 to send the request. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values.
- *u8DestinationEndPointId*: Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address type eZCL_AMBOUND.
- *psDestinationAddress*: Pointer to a structure holding the address of the node to which the request is sent.
- *pu8TransactionSequenceNumber*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the request.
- *psPayload*: Pointer to a structure containing the payload for this message (see Section 13.7.2)

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

13.6.7 eCLD_ScenesCommandRemoveSceneRequestSend

```
teZCL_Status eCLD_ScenesCommandRemoveSceneRequestSend(
         uint8 u8SourceEndPointId,
         uint8 u8DestinationEndPointId,
         tsZCL_Address *psDestinationAddress,
         uint8 *pu8TransactionSequenceNumber,
        tsCLD_ScenesRemoveSceneRequestPayload
 *psPayload);
```

Description

This function sends a Remove Scene command to request that the target device deletes membership of the destination endpoint from a particular scene - that is, remove the scene from the Scene table. The relevant scene ID is specified in the payload of the message. The scene may also be associated with a particular group.

The device receiving this message generates a callback event on the endpoint on which the Scenes cluster was registered. The device then deletes the scene in the Scene table. If the request was sent to a single device (rather than to a group address), it then generates an appropriate Remove Scene response indicating success, or failure (see Section 13.7.3).

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is 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 to send the request. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values.
- *u8DestinationEndPointId*: Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP.
- *psDestinationAddress*: Pointer to a structure holding the address of the node to which the request is sent.
- *pu8TransactionSequenceNumber*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the request.
- *psPayload*: Pointer to a structure containing the payload for this message (see Section 13.7.2).

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

13.6.8 eCLD_ScenesCommandRemoveAllScenesRequestSend

```
teZCL_Status eCLD_ScenesCommandRemoveAllScenesRequestSend(
         uint8 u8SourceEndPointId,
         uint8 u8DestinationEndPointId,
         tsZCL_Address *psDestinationAddress,
         uint8 *pu8TransactionSequenceNumber,
         tsCLD_ScenesRemoveAllScenesRequestPayload
 *psPayload);
```

Description

This function sends a Remove All Scenes command to request that the target device deletes all entries corresponding to the specified group ID/address in its Scene table. The relevant group ID is specified in the payload of the message. Note that specifying a group ID of 0x0000 removes all scenes not associated with a group.

The device receiving this message generates a callback event on the endpoint on which the Scenes cluster was registered. The device then deletes the scenes in the Scene table. If the request is sent to a single device (rather than to a group address), it then generates an appropriate Remove All Scenes response indicating success, or failure (see Section 13.7.3).

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is 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 to send the request. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values.
- *u8DestinationEndPointId*: Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP.
- *psDestinationAddress*: Pointer to a structure holding the address of the node to which the request is sent.
- *pu8TransactionSequenceNumber*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the request.
- *psPayload*: Pointer to a structure containing the payload for this message (see Section 13.7.2).

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

13.6.9 eCLD_ScenesCommandStoreSceneRequestSend

```
teZCL_Status eCLD_ScenesCommandStoreSceneRequestSend(
         uint8 u8SourceEndPointId,
         uint8 u8DestinationEndPointId,
         tsZCL_Address *psDestinationAddress,
         uint8 *pu8TransactionSequenceNumber,
         tsCLD_ScenesStoreSceneRequestPayload
 *psPayload);
```

Description

This function sends a Store Scene command to request that the target device saves the current settings of all other clusters on the device as a scene - that is, adds a scene containing the current cluster settings to the Scene table. The entry is stored using the scene ID and group ID specified in the payload of the command. If an entry exists with these IDs, its existing cluster settings are overwritten with the new settings.

Note that this command does not set the transition time and scene name fields (or for a new entry, they are set to null values). If this command is to create a new scene that requires particular settings for these fields, Add Group command should be used. The scene entry must be created in advance using the Add Group command, and the fields transition time and scene name should be pre-configured.

The device receiving this message generates a callback event on the endpoint on which the Scenes cluster was registered. If the request is sent to a single device (rather than to a group address), it then generates an appropriate Store Scene response indicating success, or failure (see Section 13.7.3).

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is 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*: ID or number of the local endpoint through which to send the request. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values.
- *u8DestinationEndPointId*: ID or number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP.
- *psDestinationAddress*: Pointer to a structure holding the address of the node to which the request is sent.
- *pu8TransactionSequenceNumber*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the request.
- *psPayload*: Pointer to a structure containing the payload for this message (see Section 13.7.2).

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_F

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

13.6.10 eCLD_ScenesCommandRecallSceneRequestSend

```
teZCL_Status eCLD_ScenesCommandRecallSceneRequestSend(
         uint8 u8SourceEndPointId,
         uint8 u8DestinationEndPointId,
         tsZCL_Address *psDestinationAddress,
         uint8 *pu8TransactionSequenceNumber,
         tsCLD_ScenesRecallSceneRequestPayload
 *psPayload);
```

Description

This function sends a Recall Scene command to request that the target device retrieves and implements the settings of the specified scene - that is, reads the scene settings from the Scene table and applies them to the other clusters on the device. The required scene ID and group ID are specified in the payload of the command.

Note that if the specified scene entry does not contain any settings for a particular cluster or there are some missing attribute values for a cluster, these attribute values remains unchanged in the implementation of the cluster.

The device receiving this message generates a callback event on the endpoint on which the Scenes cluster was registered. If the request is sent to a single device (rather than to a group address), it then generates an appropriate Recall Scene response indicating success, or failure (see Section 13.7.3).

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is 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 to send the request. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values.
- *u8DestinationEndPointId*: Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP.
- *psDestinationAddress*: Pointer to a structure holding the address of the node to which the request is sent.
- *pu8TransactionSequenceNumber*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the request.
- *psPayload*: Pointer to a structure containing the payload for this message (see Section 13.7.2).

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

13.6.11 eCLD_ScenesCommandGetSceneMembershipRequestSend

```
teZCL_Status eCLD_ScenesCommandGetSceneMembershipRequestSend(
         uint8 u8SourceEndPointId,
         uint8 u8DestinationEndPointId,
         tsZCL_Address *psDestinationAddress,
         uint8 *pu8TransactionSequenceNumber,
         tsCLD_ScenesGetSceneMembershipRequestPayload
 *psPayload);
```

Description

This function sends a Get Scene Membership to inquire which scenes are associated with a specified group ID on a device. The relevant group ID is specified in the payload of the command.

The device receiving this message generates a callback event on the endpoint on which the Scenes cluster is registered. If the request is sent to a single device (rather than to a group address), it then generates an appropriate Get Scene Membership response indicating success or failure and, if successful, the response contains a list of the scene IDs associated with the given group ID (see Section 13.7.3). If the original command

is sent to a group address, an individual device only responds if it has scenes associated with the group ID in the command payload (so it only responds if successful).

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is 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 to send the request. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values.
- *u8DestinationEndPointId*: Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP.
- *psDestinationAddress*: Pointer to a structure holding the address of the node to which the request is sent.
- *pu8TransactionSequenceNumber*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the request.
- *psPayload*: Pointer to a structure containing the payload for this message (see Section 13.7.2).

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

13.6.12 eCLD_ScenesCommandEnhancedAddSceneRequestSend

```
teZCL_Status eCLD_ScenesCommandEnhancedAddSceneRequestSend(
        uint8 u8SourceEndPointId,
        uint8 u8DestinationEndPointId,
        tsZCL_Address *psDestinationAddress,
        uint8 *pu8TransactionSequenceNumber,
        tsCLD_ScenesEnhancedAddSceneRequestPayload
 *psPayload);
```

Description

This function sends an Enhanced Add Scene command to a remote device in order to add a scene on the specified endpoint - that is, to add an entry to the Scene table on the endpoint. The function allows a finer transition time (in tenths of a second rather than seconds) when applying the scene. The scene ID is specified in the payload of the message, along with a duration for the scene and the transition time, among other values (see Section 13.7.2). The scene may also be associated with a particular group.

The device receiving this message generates a callback event on the endpoint on which the Scenes cluster is registered. If possible, add the scene to its Scene table before sending an Enhanced Add Scene response indicating success or failure (see Section 13.7.3).

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is 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 to send the request. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values.
- *u8DestinationEndPointId*: Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP.
- *psDestinationAddress*: Pointer to a structure holding the address of the node to which the request is sent.
- *pu8TransactionSequenceNumber*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the request.
- *psPayload*: Pointer to a structure containing the payload for this message (see Section 13.7.2).

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

13.6.13 eCLD_ScenesCommandEnhancedViewSceneRequestSend

```
teZCL_Status eCLD_ScenesCommandEnhancedViewSceneRequestSend(
        uint8 u8SourceEndPointId,
        uint8 u8DestinationEndPointId,
        tsZCL_Address *psDestinationAddress,
        uint8 *pu8TransactionSequenceNumber,
        tsCLD_ScenesEnhancedViewSceneRequestPayload
 *psPayload);
```

Description

This function sends an Enhanced View Scene command to a remote device, requesting information on a particular scene on the destination endpoint. The returned information includes the finer transition time configured with the function **eCLD_ScenesCommandEnhancedAddSceneRequestSend()**. The relevant scene ID is specified in the command payload. Note that this command can only be sent to an individual device/endpoint and not to a group address.

The device receiving this message generates a callback event on the endpoint on which the Scenes cluster was registered and generates an Enhanced View Scene response containing the relevant information (see Section 13.7.3).

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is 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 to send the request. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values.
- *u8DestinationEndPointId*: Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address type eZCL_AMBOUND.
- *psDestinationAddress*: Pointer to a structure holding the address of the node to which the request is sent.
- *pu8TransactionSequenceNumber*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the request.
- *psPayload*: Pointer to a structure containing the payload for this message (see Section 13.7.2)

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

13.6.14 eCLD_ScenesCommandCopySceneSceneRequestSend

```
teZCL_Status eCLD_ScenesCommandCopySceneSceneRequestSend(
        uint8 u8SourceEndPointId,
        uint8 u8DestinationEndPointId,
        tsZCL_Address *psDestinationAddress,
        uint8 *pu8TransactionSequenceNumber,
        tsCLD_ScenesCopySceneRequestPayload *psPayload);
```

Description

This function sends a Copy Scene command to a remote device, requesting that the scene settings from one scene ID/group ID combination are copied to another scene ID/group ID combination on the target endpoint. The relevant source and destination scene ID/group ID combinations are specified in the command payload.

Note that:

- If the destination scene ID/group ID exists on the target endpoint, the existing scene is overwritten with the new settings.
- The message payload contains a ‘copy all scenes’ bit. If the bit is set to ‘1’, it instructs the destination server to copy all scenes in the specified source group to scenes with the same scene IDs in the destination group. In this case, the source and destination scene IDs in the payload are ignored.

The device receiving this message generates a callback event on the endpoint on which the Scenes cluster was registered and, if the original request is unicast, generates a Copy Scene response (see Section 13.7.3).

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is 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 to send the request. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values.
- *u8DestinationEndPointId*: Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address type eZCL_AMBOUND.
- *psDestinationAddress*: Pointer to a structure holding the address of the node to which the request is sent.
- *pu8TransactionSequenceNumber*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the request.
- *psPayload*: Pointer to a structure containing the payload for this message (see Section 13.7.2).

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

13.7 Structures

13.7.1 Custom Data Structure

The Scenes cluster requires extra storage space to be allocated to be used by internal functions. The structure definition for this storage is shown below:

```
typedef struct
{
    DLIST   lScenesAllocList;
    DLIST   lScenesDeAllocList;
    tsZCL_ReceiveEventAddress       sReceiveEventAddress;
    tsZCL_CallBackEvent             sCustomCallBackEvent;
    tsCLD_ScenesCallBackMessage     sCallBackMessage;
    tsCLD_ScenesTableEntry
          asScenesTableEntry[CLD_SCENES_MAX_NUMBER_OF_SCENES];
} tsCLD_ScenesCustomDataStructure;
```

The fields are for internal use and no knowledge of them is required.

13.7.2 Custom Command Payloads

The following structures contain the payloads for the Scenes cluster custom commands.

Add Scene Request Payload

```
typedef struct
{
    uint16                      u16GroupId;
    uint8                       u8SceneId;
    uint16                      u16TransitionTime;
    tsZCL_CharacterString       sSceneName;
    tsCLD_ScenesExtensionField  sExtensionField;
} tsCLD_ScenesAddSceneRequestPayload;
```

where:

- `u16GroupId` is the group ID with which the scene is associated (0x0000 if there is no association with a group)
- `u8SceneId` is the ID of the scene to be added to the Scene table (the Scene ID must be unique within the group associated with the scene)
- `u16TransitionTime` is the amount of time, in seconds, that the device takes to switch to this scene
- `sSceneName` is an optional character string (of up to 16 characters) representing the name of the scene
- `sExtensionField` is a structure containing the attribute values of the clusters to which the scene relates

View Scene Request Payload

```
typedef struct
{
    uint16                      u16GroupId;
    uint8                       u8SceneId;
} tsCLD_ScenesViewSceneRequestPayload;
```

where:

- `u16GroupId` is the group ID with which the desired scene is associated
- `u8SceneId` is the scene ID of the scene to be viewed

Remove Scene Request Payload

```
typedef struct
{
    uint16                      u16GroupId;
    uint8                       u8SceneId;
} tsCLD_ScenesRemoveSceneRequestPayload;
```

where:

- `u16GroupId` is the group ID with which the relevant scene is associated
- `u8SceneId` is the scene ID of the scene to be deleted from the Scene table

Remove All Scenes Request Payload

```
typedef struct
```

```
{
    uint16                      u16GroupId;
} tsCLD_ScenesRemoveAllScenesRequestPayload;
```

where `u16GroupId` is the group ID for which all scenes are to be deleted.

Store Scene Request Payload

```
typedef struct
{
    uint16                      u16GroupId;
    uint8                       u8SceneId;
} tsCLD_ScenesStoreSceneRequestPayload;
```

where:

- `u16GroupId` is the group ID with which the relevant scene is associated
- `u8SceneId` is the scene ID of the scene in which the captured cluster settings are to be stored

Recall Scene Request Payload

```
typedef struct
{
    uint16                      u16GroupId;
    uint8                       u8SceneId;
} tsCLD_ScenesRecallSceneRequestPayload;
```

where:

- `u16GroupId` is the group ID with which the relevant scene is associated
- `u8SceneId` is the scene ID of the scene from which cluster settings are to be retrieved and applied

Get Scene Membership Request Payload

```
typedef struct
{
    uint16                      u16GroupId;
} tsCLD_ScenesGetSceneMembershipRequestPayload;
```

where `u16GroupId` is the group ID for which associated scenes are required.

Enhanced Add Scene Request Payload

```
typedef struct
{
    uint16                      u16GroupId;
    uint8                       u8SceneId;
    uint16                      u16TransitionTime100ms;
    tsZCL_CharacterString       sSceneName;
    tsCLD_ScenesExtensionField  sExtensionField;
} tsCLD_ScenesEnhancedAddSceneRequestPayload;
```

where:

- u16GroupId is the group ID with which the scene is associated (0x0000 if there is no association with a group)
- u8SceneId is the ID of the scene to be added to the Scene table (the Scene ID must be unique within the group associated with the scene)
- u16TransitionTime100ms is the amount of time, in tenths of a second, that the device takes to switch to this scene
- sSceneName is an optional character string (of up to 16 characters) representing the name of the scene
- sExtensionField is a structure containing the attribute values of the clusters to which the scene relates

View Scene Request Payload

```
typedef struct
{
    uint16                      u16GroupId;
    uint8                       u8SceneId;
} tsCLD_ScenesEnhancedViewSceneRequestPayload;
```

where:

- u16GroupId is the group ID with which the desired scene is associated
- u8SceneId is the scene ID of the scene to be viewed

Copy Scene Request Payload

```
typedef struct
{
    uint8       u8Mode;
    uint16      u16FromGroupId;
    uint8       u8FromSceneId;
    uint16      u16ToGroupId;
    uint8       u8ToSceneId;
} tsCLD_ScenesCopySceneRequestPayload;
```

where:

- u8Mode is a bitmap indicating the required copying mode (only bit 0 is used):
  - If bit 0 is set to ‘1’, then ‘copy all scenes’ mode is used, in which all scenes associated with the source group are duplicated for the destination group (and the scene ID fields are ignored)
  - If bit 0 is set to ‘0’, then a single scene is copied
- u16FromGroupId is the source group ID
- u8FromSceneId is the source scene ID (ignored for ‘copy all scenes’ mode)
- u16ToGroupId is the destination group ID
- u8ToSceneId is the destination scene ID (ignored for ‘copy all scenes’ mode)

13.7.3 Custom Command Responses

The Scenes cluster generates responses to certain custom commands. The responses which contain payloads are detailed below:

Add Scene Response Payload

```
typedef struct
{
```

```
    zenum8                      eStatus;
    uint16                      u16GroupId;
    uint8                       u8SceneId;
} tsCLD_ScenesAddSceneResponsePayload;
```

where:

- `eStatus` is the outcome of the Add Scene command (success or invalid)
- `u16GroupId` is the group ID with which the added scene is associated
- `u8SceneId` is the scene ID of the added scene

View Scene Response Payload

```
typedef struct
{
    zenum8                      eStatus;
    uint16                      u16GroupId;
    uint8                       u8SceneId;
    uint16                      u16TransitionTime;
    tsZCL_CharacterString       sSceneName;
    tsCLD_ScenesExtensionField  sExtensionField;
} tsCLD_ScenesViewSceneResponsePayload;
```

where:

- `eStatus` is the outcome of the View Scene command (success or invalid)
- `u16GroupId` is the group ID with which the viewed scene is associated
- `u8SceneId` is the scene ID of the viewed scene
- `u16TransitionTime` is the amount of time, in seconds, that the device takes to switch to the viewed scene
- `sSceneName` is an optional character string (of up to 16 characters) representing the name of the viewed scene
- `sExtensionField` is a structure containing the attribute values of the clusters to which the viewed scene relates

Remove Scene Response Payload

```
typedef struct
{
    zenum8                      eStatus;
    uint16                      u16GroupId;
    uint8                       u8SceneId;
} tsCLD_ScenesRemoveSceneResponsePayload;
```

where:

- `eStatus` is the outcome of the Remove Scene command (success or invalid)
- `u16GroupId` is the group ID with which the removed scene is associated
- `u8SceneId` is the scene ID of the removed scene

Remove All Scenes Response Payload

```
typedef struct
{
    zenum8                      eStatus;
    uint16                      u16GroupId;
```

```
} tsCLD_ScenesRemoveAllScenesResponsePayload;
```

where:

- `eStatus` is the outcome of the Remove All Scenes command (success or invalid)
- `u16GroupId` is the group ID with which the removed scenes are associated

Store Scene Response Payload

```
typedef struct
{
    zenum8                      eStatus;
    uint16                      u16GroupId;
    uint8                       u8SceneId;
} tsCLD_ScenesStoreSceneResponsePayload;
```

where:

- `eStatus` is the outcome of the Store Scene command (success or invalid)
- `u16GroupId` is the group ID with which the stored scene is associated
- `u8SceneId` is the scene ID of the stored scene

Get Scene Membership Response Payload

```
typedef struct
{
    zenum8                      eStatus;
    uint8                       u8Capacity;
    uint16                      u16GroupId;
    uint8                       u8SceneCount;
    uint8                       *pu8SceneList;
} tsCLD_ScenesGetSceneMembershipResponsePayload;
```

where:

- `eStatus` is the outcome of the Get Scene Membership command (success or invalid)
- `u8Capacity` is the capacity of the Scene table of the device to receive more scenes - that is, the number of scenes that may be added (special values: 0xFE means that at least one more scene may be added, a higher value means that the remaining capacity of the table is unknown)
- `u16GroupId` is the group ID to which the query relates
- `u8SceneCount` is the number of scenes in the list of the next field
- `pu8SceneList` is a pointer to the returned list of scenes from those queried that exist on the device, where each scene is represented by its scene ID

Enhanced Add Scene Response Payload

```
typedef struct
{
    zenum8     eStatus;
    uint16     u16GroupId;
    uint8      u8SceneId;
} tsCLD_ScenesEnhancedAddSceneResponsePayload;
```

where:

- eStatus is the outcome of the Enhanced Add Scene command (success or invalid)
- u16GroupId is the group ID with which the added scene is associated
- u8SceneId is the scene ID of the added scene

Enhanced View Scene Response Payload

```
typedef struct
{
    zenum8                      eStatus;
    uint16                      u16GroupId;
    uint8                       u8SceneId;
    uint16                      u16TransitionTime;
    tsZCL_CharacterString       sSceneName;
    tsCLD_ScenesExtensionField  sExtensionField;
} tsCLD_ScenesEnhancedViewSceneResponsePayload;
```

where:

- eStatus is the outcome of the Enhanced View Scene command (success or invalid)
- u16GroupId is the group ID with which the viewed scene is associated
- u8SceneId is the scene ID of the viewed scene
- u16TransitionTime is the amount of time, in seconds, that the device takes to switch to the viewed scene
- sSceneName is an optional character string (of up to 16 characters) representing the name of the viewed scene
- sExtensionField is a structure containing the attribute values of the clusters to which the viewed scene relates

Copy Scene Response Payload

```
typedef struct
{
    uint8       u8Status;
    uint16      u16FromGroupId;
    uint8       u8FromSceneId;
} tsCLD_ScenesCopySceneResponsePayload;
```

where:

- u8Status is the outcome of the Copy Scene command (success, invalid scene, or insufficient space for new scene)
- u16FromGroupId is the source group ID for the copy
- u8FromSceneId is the source scene ID for the copy

13.7.4 Scenes Table Entry

The following structure contains the data for a Scenes table entry (containing a saved scene):

```
typedef struct
{
    DNODE           dllScenesNode;
    bool            bActive;
    bool            bInTransit;
    uint16          u16GroupId;
    uint8           u8SceneId;
    uint16          u16TransitionTime;
```

```
    uint32          u32TransitionTimer;
    uint8           u8SceneNameLength;
    uint8           au8SceneName[CLD_SCENES_MAX_SCENE_NAME_LENGTH + 1];
    uint16          u16SceneDataLength;
    uint8           au8SceneData[CLD_SCENES_MAX_SCENE_STORAGE_BYTES];
#ifdef  CLD_SCENES_TABLE_SUPPORT_TRANSITION_TIME_IN_MS
    uint8           u8TransitionTime100ms;
#endif
} tsCLD_ScenesTableEntry;
```

where:

- `bActive` is a boolean value indicating whether the scene is active (TRUE) or inactive (FALSE).
- `bInTransit` is a boolean value indicating whether the scene is in a transitional state (TRUE) or a constant active/inactive state (FALSE).
- `u16GroupId` is the identifier of the group to which the scene applies (the value 0x0000 is used to indicate that the scene is not associated with a group).
- `u8SceneId` is the identifier of the scene and must be a unique value within the group with which the scene is associated.
- `u16TransitionTime` is the length of time, in seconds, that the device takes to move from its current state to the scene state.
- `u32TransitionTimer` is the elapsed time, in milliseconds, since the start of a currently active transition to the scene.
- `u8SceneNameLength` is the number of characters in the name of the scene (and therefore the size of the array `au8SceneName[]` below). The value must not be greater than CLD_SCENES_MAX_SCENE_NAME_LENGTH, which is set in the compile-time options (see Section 13.9).
- `au8SceneName[]` is an array containing the name of the scene, with one ASCII character in each array element. The number of elements in the array must be set in `u8SceneNameLength` above.
- `u16SceneDataLength` is the number of items of data for the scene (and therefore the size of the array `au8SceneData[]` below). The value must not be greater than CLD_SCENES_MAX_SCENE_STORAGE_BYTES, which is set in the compile-time options (see Section 13.9).
- `au8SceneData[]` is an array containing the data for the scene, with one data item in each array element. The data stored is dependent on the cluster to which the scene data applies. The number of elements in the array must be set in `u16SceneDataLength` above.
- `u8TransitionTime` 100 ms is an optional that allows a fractional part to be added to the transition time (`u16TransitionTime`) of the scene. This value represents the number of tenths of a second in the range 0x00 to 0x09.

13.8 Enumerations

13.8.1 teCLD_Scenes_ClusterID

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

```
typedef enum
{
    E_CLD_SCENES_ATTR_ID_SCENE_COUNT            = 0x0000,   /* Mandatory */
    E_CLD_SCENES_ATTR_ID_CURRENT_SCENE,                     /* Mandatory */
    E_CLD_SCENES_ATTR_ID_CURRENT_GROUP,                     /* Mandatory */
    E_CLD_SCENES_ATTR_ID_SCENE_VALID,                       /* Mandatory */
    E_CLD_SCENES_ATTR_ID_NAME_SUPPORT,                      /* Mandatory */
    E_CLD_SCENES_ATTR_ID_LAST_CONFIGURED_BY                 /* Optional  */
} teCLD_Scenes_ClusterID;
```

13.9 Compile-time options

To enable the Scenes cluster in the code to be built, it is necessary to add the following to the **zcl_options.h** file:

```
#define CLD_SCENES
```

In addition, to include the software for a cluster client or server or both, it is necessary to add one or both of the following to the same file:

```
#define SCENES_CLIENT
#define SCENES_SERVER
```

The Scenes cluster contains macros that may be optionally specified at compile time by adding some or all the following lines to the **zcl_options.h** file.

To enable the optional Last Configured By attribute, add this line:

```
#define CLD_SCENES_ATTR_LAST_CONFIGURED_BY
```

To configure the maximum length of the Scene Name storage, add this line:

```
#define CLD_SCENES_MAX_SCENE_NAME_LENGTH            (16)
```

To configure the maximum number of scenes, add this line:

```
#define CLD_SCENES_MAX_NUMBER_OF_SCENES             (16)
```

To configure the maximum number of bytes available for scene storage, add this line:

```
#define CLD_SCENES_MAX_SCENE_STORAGE_BYTES          (20)
```

To enable the Enhanced Add Scene command, add this line:

```
#define CLD_SCENES_CMD_ENHANCED_ADD_SCENE
```

To enable the Enhanced View Scene command, add this line:

```
#define CLD_SCENES_CMD_ENHANCED_VIEW_SCENE
```

To enable the Copy Scene command, add this line:

```
#define CLD_SCENES_CMD_COPY_SCENE
```

To enable TransitionTime100ms in Scene tables, add this line:

```
#define CLD_SCENES_TABLE_SUPPORT_TRANSITION_TIME_IN_MS
```

To define the value (n) of the Cluster Revision attribute, add this line:

```
#define CLD_SCENES_CLUSTER_REVISION <n>
```

The default value is 1, which corresponds to the revision of the cluster in the ZCL r6 specification (see Section 2.4).

14 On/Off Cluster

This chapter describes the On/Off cluster.

The On/Off cluster has a Cluster ID of 0x0006.

14.1 Overview

The On/Off cluster allows a device to be put into the 'on' and 'off' states, or toggled between the two states. The cluster also provides the following enhanced functionality for lighting:

- When switching off lights with an effect, saves the last light (attribute) settings to a global scene, ready to be reused for the next switch-on from the global scene - see Section 14.5.2 and Section 14.6
- Allows lights to be switched on for a timed period (and then automatically switched off) - see Section 14.5.3

To use the functionality of this cluster, you must include the file **OnOff.h** in your application and enable the cluster by defining CLD_ONOFF in the **zcl_options.h** file.

It is also necessary to enable the cluster as a server or client, or as both:

- The cluster server is able to receive commands to change the on/off state of the local device.
- The cluster client is able to send commands to the server to request a change to the on/off state of the remote device.

The inclusion of the client or server software must be pre-defined in the compile-time options of the application. In addition, if the cluster is to reside on a custom endpoint then the role of client or server must also be specified when creating the cluster instance.

The compile-time options for the On/Off cluster are fully detailed in Section 14.10.

14.2 On/Off Cluster Structure and Attribute

The structure definition for the On/Off cluster is:

```
typedef struct
{
#ifdef ONOFF_SERVER
    zbool                   bOnOff;
#ifdef CLD_ONOFF_ATTR_GLOBAL_SCENE_CONTROL
    zbool                   bGlobalSceneControl;
#endif
#ifdef CLD_ONOFF_ATTR_ON_TIME
    zuint16                 u16OnTime;
#endif
#ifdef CLD_ONOFF_ATTR_OFF_WAIT_TIME
    zuint16                 u16OffWaitTime;
#endif
#ifdef CLD_ONOFF_ATTR_STARTUP_ONOFF
    /* ZLO extension for OnOff Cluster    */
    zenum8                  eStartUpOnOff;
#endif
#ifdef CLD_ONOFF_ATTR_ATTRIBUTE_REPORTING_STATUS
    zenum8                  u8AttributeReportingStatus;
#endif
#endif
    zuint16                 u16ClusterRevision;
} tsCLD_OnOff;
```

where:

- `bOnOff` is the on/off state of the device (TRUE = on, FALSE = off).
- `bGlobalSceneControl` is an optional attribute for lighting that is used with the global scene - the value of this attribute determines whether to permit saving the current light settings to the global scene:
  - TRUE - Current light settings can be saved to the global scene
  - FALSE - Current light settings cannot be saved to the global scene
- `u16OnTime` is an optional attribute for lighting used to store the time, in tenths of a second, for which the lights remain ‘on’ after a switch-on with ‘timed off’
  (that is, the time before starting the transition from the ‘on’ state to the ‘off’ state). The special values 0x0000 and 0xFFFF indicate that the lamp must be maintained in the ‘on’ state indefinitely (no timed off).
- `u16OffWaitTime` is an optional attribute for lighting used to store the waiting time, in tenths of a second, following a ‘timed off’ before the lights can be again switched on with a ‘timed off’.

Note: *If the `bGlobalSceneControl` attribute and global scene are to be used, the Scenes and Groups clusters must also be enabled - see Chapter 13 and Chapter 12.*

- `eStartUpOnOff` is an optional attribute that is used in the lighting domain to define the required start-up behavior of a light device when it is supplied with power. It determines the initial value of `bOnOff` on start-up. The possible values and behaviors are as follows:

Table 31. eStartUpOnOff attribute

eStartUpOnOff	Behavior
0x00	Put the light in the off state - set `bOnOff` to FALSE
0x01	Put the light in the on state - set `bOnOff` to TRUE
0x02	Toggle the light from its previous state: • If `bOnOff` was FALSE, set it to TRUE • If `bOnOff` was TRUE, set it to FALSE
0x03-0xFE	Reserved
0xFF	Put the light in its previous state - set `bOnOff` to its previous value

- `u8AttributeReportingStatus` is an optional attribute that should be enabled when attribute reporting is used for the cluster (see Section 2.3.5). The value of this attribute indicates whether there are attribute reports still pending (0x00) or the attribute reports are complete (0x01) - all other values are reserved. This attribute is also described in Section 2.4.
- `u16ClusterRevision` is a mandatory attribute that specifies the revision of the cluster specification on which this cluster instance is based. The cluster specification in the ZCL r6 corresponds to a cluster revision of 1. The value is incremented by one for each subsequent revision of the cluster specification. This attribute is also described in Section 2.4.

14.3 Attributes for Default Reporting

The following attribute of the On/Off cluster can be selected for default reporting:

- `bOnOff`

Attribute reporting (including default reporting) is described in Appendix B. Enabling reports for this attribute is described in Appendix B.3.6.

14.4 Initialization

The function **eCLD_OnOffCreateOnOff()** is used to create an instance of the On/Off cluster. The function is called by the initialization function for the host device.

Note: *If the global scene is to be used to remember light settings, then. Scenes and Groups cluster instances must be created - see Chapter 13 and Chapter 12.*

14.5 Sending Commands

The NXP implementation of the ZCL provides functions for sending commands between an On/Off cluster client and server. A command is sent from the client to one or more endpoints on the server. Multiple endpoints can usually be targeted using binding or group addressing.

14.5.1 Switching On and Off

A remote device (supporting the On/Off cluster server) can be switched on, switched off or, toggled between the on and off states by calling the function **eCLD_OnOffCommandSend()** on a cluster client. In the case of a toggle, if the device is initially in the on state it is switched off and if the device is initially in the off state it is switched on.

14.5.1.1 Timeout on the ‘On’ Command

On receiving an 'On' command, a timeout is applied such that the 'on' state is maintained for a specified duration before automatically switching to the 'off' state. This timeout is defined using the optional attributes `u16OnConfigurableDuration` and `eDurationUnitOfMeasurement`. The timeout duration in seconds is given by:

`u16OnConfigurableDuration` * 10^(power from `eDurationUnitOfMeasurement`)

The attribute `u16OnConfigurableDuration` can be set locally or remotely, while the attribute `eDurationUnitOfMeasurement` must be set locally. A maximum timeout duration can be defined locally via the optional attribute `u16MaxDuration`, which puts an upper limit on the value of `u16OnConfigurableDuration`.

The attribute `u16OnConfigurableDuration` can be set remotely using the **eZCL_SendWriteAttributesRequest()** function. On receiving this write request, the local ZCL checks that the requested duration is within the permissible range (see Section 2.3.3.1) - if the request exceeds the maximum permitted value, the timeout duration is clipped to this maximum.

For full details of the above attributes, refer to Section 14.2.

When an ‘On’ command is received, an E_ZCL_CBET_CLUSTER_CUSTOM event is generated. The application is responsible for implementing the timeout described above, if it is enabled. First, the application must check the attributes `u16OnConfigurableDuration` and `eDurationUnitOfMeasurement` to make sure they have valid values. If so, the application must start a timer to implement the timeout for the duration defined by these attributes. On expiration of the timer, the application must switch from the ‘on’ state to the ‘off’ state by (locally) writing to the `bOnOff` attribute.

14.5.1.2 On/Off with Transition Effect

If the Level Control cluster (see Chapter 16) is also used on the target device, an ‘On’ or ‘Off’ command can be implemented with a transition effect, as follows:

- If the optional Level Control ‘On Transition Time’ attribute is enabled, an ‘On’ command results in a gradual transition. This transition is from the ‘off’ level to the ‘on’ level over the time-interval specified by the attribute.
- If the optional Level Control ‘Off Transition Time’ attribute is enabled, an ‘Off’ command results in a gradual transition from the ‘on’ level to the ‘off’ level over the time-interval specified by the attribute.

14.5.2 Switching Off Lights with Effect

In the case of lighting, lights can be (remotely) switched off with an effect by calling the function **eCLD_OnOffCommandOffWithEffectSend()** on an On/Off cluster client.

Two ‘off effects’ are available and there are variants of each effect:

- **Fade**, with the following variants:
  - Fade to off in 0.8 seconds (default)
  - Reduce brightness by 50 % in 0.8 seconds then fade to off in 4 seconds
  - No fade
- **Rise and fall**, with (currently) only one variant:
  - Increase brightness by 20 % (if possible) in 0.5 seconds then fade to off in 1 second (default)

14.5.3 Switching On Timed Lights

In the case of lighting, lights can be switched on temporarily and automatically switched off at the end of a timed period. This kind of switch-on can be initiated remotely using the function **CLD_OnOffCommandOnWithTimedOffSend()** on an On/Off cluster client. In addition, a waiting time can be implemented after the automatic switch-off, during which the lights cannot be switched on again using the above function (although a normal switch-on is possible).

The following values must be specified:

- Time for which the lights remain on (in tenths of a second)
- Waiting time following the automatic switch-off (in tenths of a second)

In addition, the circumstances in which the command can be accepted must be specified - that is, accepted at any time (except during the waiting time) or only when the lights are already on. The latter case can be used to initiate a timed switch-off.

14.6 Saving Light Settings

In the case of lighting, the current light (attribute) settings can be automatically saved to a ‘global scene’ when switching off the lights using the function **eCLD_OnOffCommandOffWithEffectSend()**. If the lights are, then switched on with the E_CLD_ONOFF_CMD_ON_RECALL_GLOBAL_SCENE option in **eCLD_OnOffCommandSend()**, the saved light settings are reloaded. In this way, the system remembers the last light settings used before switch-off and resumes with these settings at the next switch-on. This feature is useful when the light levels are adjustable using the Level Control cluster (Chapter 16) and/or the light colours are adjustable using the Colour Control cluster (Chapter 31).

The attribute values corresponding to the current light settings are saved (locally) to a global scene with scene ID and group ID both equal to zero. Therefore, to use this feature:

- Scenes cluster must be enabled and a cluster instance created
- Groups cluster must be enabled and a cluster instance created
- Optional On/Off cluster attribute `bGlobalSceneControl` must be enabled

The above attribute is a boolean which determines whether to permit the current light settings to be saved to the global scene. The attribute is set to FALSE after a switch-off using the function **eCLD_OnOffCommandOffWithEffectSend()**. It is set to TRUE after a switch-on or a change in the light settings (attributes) - more specifically, after a change resulting from a Level Control cluster ‘Move to Level with On/Off’ command, from a Scenes cluster ‘Recall Scene’ command, or from an On/Off cluster 'On' command or 'On With Recall Global Scene' command.

14.7 Functions

The following On/Off cluster functions are provided in the NXP implementation of the ZCL:

1. eCLD_OnOffCreateOnOff
2. _OnOffCommandSend
3. eCLD_OnOffCommandOffWithEffectSend
4. eCLD_OnOffCommandOnWithTimedOffSend

14.7.1 eCLD_OnOffCreateOnOff

```
teZCL_Status eCLD_OnOffCreateOnOff(
         tsZCL_ClusterInstance *psClusterInstance,
         bool_t bIsServer,
         tsZCL_ClusterDefinition *psClusterDefinition,
         void *pvEndPointSharedStructPtr,
         uint8 *pu8AttributeControlBits,
        tsCLD_OnOffCustomDataStructure
 *psCustomDataStructure);
```

Description

This function creates an instance of the On/Off 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 must 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 creates an On/Off cluster instance on the endpoint, but instances of other clusters may also be created on the same endpoint by calling their corresponding creation functions.

Note: *Do not call this function for an endpoint on which a standard ZigBee device is used. In this case, the device and its supported clusters must be registered on the endpoint using the relevant device registration function.*

When used, this function must be the first On/Off cluster function called in the application, and must be called after the stack has been started and after the ZCL has been initialized.

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 On/Off cluster. The function initialises the array elements to zero.

Parameters

- *psClusterInstance*: Pointer to structure containing information about the cluster instance to be created (see Section 6.1.16). This structure is updated by the function by initialising individual structure fields.
- *bIsServer*: Type of cluster instance (server or client) to be created:
  - TRUE - server
  - FALSE - client
- *psClusterDefinition*: Pointer to structure indicating the type of cluster to be created (see Section 6.1.2). In this case, this structure must contain the details of the On/Off cluster. This parameter can refer to a pre-filled structure called `sCLD_OnOff` which is provided in the **OnOff.h** file.

- *pvEndPointSharedStructPtr*: Pointer to the shared structure used for attribute storage. This parameter should be the address of the structure of type `tsCLD_OnOff` which defines the attributes of On/Off cluster. The function initialises the attributes with default values.
- *pu8AttributeControlBits*: Pointer to an array of **uint8** values, with one element for each attribute in the cluster (see above)
- *psCustomDataStructure*: Pointer to a structure containing the storage for internal functions of the cluster (see Section 14.8.1)

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL

14.7.2 eCLD_OnOffCommandSend

```
teZCL_Status eCLD_OnOffCommandSend(
         uint8 u8SourceEndPointId,
         uint8 u8DestinationEndPointId,
         tsZCL_Address *psDestinationAddress,
         uint8 *pu8TransactionSequenceNumber,
         teCLD_OnOff_Command eCommand);
```

Description

This function sends a custom command instructing the target device to perform the specified operation on itself: switch off, switch on, toggle (on-to-off or off-to-on), or switch on with settings retrieved from the global scene, This last option (On with Recall Global Scene) is described in Section 14.6 and, if used, must be enabled in the compile-time options on the server (target), as indicated in Section 14.10.

The device receiving this message generates a callback event on the endpoint on which the On/Off cluster was registered.

If the Level Control cluster (see Chapter 16) is also used on the target device, an ‘On’ or ‘Off’ command can be implemented with a transition effect, as follows:

- If the optional Level Control ‘On Transition Time’ attribute is enabled, an ‘On’ command results in a gradual transition. This transition is from the ‘off’ level to the ‘on’ level over the time-interval specified in the attribute.
- If the optional Level Control ‘Off Transition Time’ attribute is enabled, an ‘Off’ command results in a gradual transition from the ‘on’ level to the ‘off’ level over the time-interval specified in the attribute.

Parameters

- *u8SourceEndPointId*: Number of the local endpoint through which to send the request. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values.
- *u8DestinationEndPointId*: Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP
- *psDestinationAddress*: Pointer to a structure holding the address of the node to which the request is sent
- *pu8TransactionSequenceNumber*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the request
- *eCommand*: Command code, one of the following:
  - E_CLD_ONOFF_CMD_OFF
  - E_CLD_ONOFF_CMD_ON

– E_CLD_ONOFF_CMD_TOGGLE
– E_CLD_ONOFF_CMD_ON_RECALL_GLOBAL_SCENE
– E_CLD_ONOFF_CMD_TOGGLE

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

14.7.3 eCLD_OnOffCommandOffWithEffectSend

```
teZCL_Status eCLD_OnOffCommandOffWithEffectSend(
        uint8 u8SourceEndPointId,
        uint8 u8DestinationEndPointId,
        tsZCL_Address *psDestinationAddress,
        uint8 *pu8TransactionSequenceNumber,
        tsCLD_OnOff_OffWithEffectRequestPayload *psPayload);
```

Description

This function sends a custom ‘Off With Effect’ command instructing the target lighting device to switch off one or more lights with the specified effect, which can be one of:

- fade (in two phases or no fade)
- rise and fall

Each of these effects is available in variants. The required effect and variant are specified in the command payload. For the payload details, refer to "Off With Effect Request Payload ".

The device receiving this message generates a callback event on the endpoint on which the On/Off cluster was registered.

Following a call to this function, the light settings on the target device are saved to a global scene, after which the attribute `bGlobalSceneControl` is set to FALSE - for more details, refer to Section 14.6.

If used, the ‘Off With Effect’ command must be enabled in the compile-time options on both the client and server, as described in Section 14.10.

Parameters

- *u8SourceEndPointId*: Number of the local endpoint through which to send the request. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values.
- *u8DestinationEndPointId*: Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types `eZCL_AMBOUND and eZCL_AMGROUP`.
- *psDestinationAddress*: Pointer to a structure holding the address of the node to which the request is sent.

- *pu8TransactionSequenceNumber*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the request.
- *psPayload*: Pointer to a structure containing the payload for this message (see Section 14.8.2).

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

14.7.4 eCLD_OnOffCommandOnWithTimedOffSend

```
teZCL_Status eCLD_OnOffCommandOnWithTimedOffSend(
        uint8 u8SourceEndPointId,
        uint8 u8DestinationEndPointId,
        tsZCL_Address *psDestinationAddress,
        uint8 *pu8TransactionSequenceNumber,
        tsCLD_OnOff_OnWithTimedOffRequestPayload
 *psPayload);
```

Description

This function sends a custom ‘On With Timed Off’ command instructing the target lighting device to switch on one or more lights for a timed period and then switch them off. In addition, a waiting time can be implemented after switch-off, during which the lights cannot be switched on again.

The following functionality must be specified in the command payload:

- Time for which the lights must remain on.
- Waiting time during which switched-off lights cannot be switched on again.
- Whether this command can be accepted at any time (outside the waiting time) or only when a light is on.

For the payload details, refer to "On With Timed Off Request Payload".

The device receiving this message generates a callback event on the endpoint on which the On/Off cluster was registered.

If used, the ‘On With Timed Off’ command must be enabled in the compile-time options on both the client and server, as described in Section 14.10.

Parameters

- *u8SourceEndPointId*: Number of the local endpoint through which to send the request. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values.
- *u8DestinationEndPointId*: Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types `eZCL_AMBOUND` and `eZCL_AMGROUP`.

- *psDestinationAddress*: Pointer to a structure holding the address of the node to which the request is sent.
- *pu8TransactionSequenceNumber*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the request.
- *psPayload*: Pointer to a structure containing the payload for this message (see Section 14.8.2).

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

14.8 Structures

14.8.1 Custom Data Structure

The On/Off cluster requires extra storage space to be allocated to be used by internal functions. The structure definition for this storage is shown below:

```
typedef struct
{
    uint8     u8Dummy;
} tsCLD_OnOffCustomDataStructure;
```

The fields are for internal use and no knowledge of them required.

14.8.2 Custom Command Payloads

Off With Effect Request Payload

```
typedef struct
{
    zuint8                  u8EffectId;
    zuint8                  u8EffectVariant;
} tsCLD_OnOff_OffWithEffectRequestPayload;
```

where:

- `u8EffectId` indicates the required ‘off effect’:
  - 0x00 - Fade
  - 0x01 - Rise and fall

    All other values are reserved.
- `u8EffectVariant` indicates the required variant of the specified ‘off effect’ - the interpretation of this field depends on the value of `u8EffectId`, as indicated in the table below.

Table 32. u8EffectId values and description

u8EffectId	u8EffectVariant	Description
0x00 (Fade)	0x00	Fade to off in 0.8 seconds (default)
	0x01	No fade
	0x02	Reduce brightness by 50 % in 0.8 seconds then fade to off in 4 seconds
	0x03-0xFF	Reserved
0x01 (Rise and fall)	0x00	Increase brightness by 20 % (if possible) in 0.5 seconds then fade to off in 1 second (default)
	0x01-0xFF	Reserved
0x02-0xFF	0x00-0xFF	Reserved

On With Timed Off Request Payload

```
typedef struct
{
    zuint8                  u8OnOff;
    zuint16                 u16OnTime;
    zuint16                 u16OffTime;
} tsCLD_OnOff_OnWithTimedOffRequestPayload;
```

where:

- `u8OnOff` indicates when the command can be accepted:
  - 0x00 - at all times (apart from in waiting time, if implemented)
  - 0x01 - only when light is on

    All other values are reserved.
- `u16OnTime` is the ‘on time’, expressed in tenths of a second in the range 0x0000 to 0xFFFE.
- `u16OffTime` is the ‘off waiting time’, expressed in tenths of a second in the range 0x0000 to 0xFFFE

14.9 Enumerations

14.9.1 teCLD_OnOff_ClusterID

The following structure contains the enumerations used to identify the attributes of the On/Off cluster.

```
typedef enum
{
    E_CLD_ONOFF_ATTR_ID_ONOFF                           = 0x0000,
    E_CLD_ONOFF_ATTR_ID_GLOBAL_SCENE_CONTROL            = 0x4000,
    E_CLD_ONOFF_ATTR_ID_ON_TIME,
    E_CLD_ONOFF_ATTR_ID_OFF_WAIT_TIME,
    #ifdef CLD_ONOFF_ATTR_STARTUP_ONOFF
    /* ZLO extension for OnOff Cluster    */
    E_CLD_ONOFF_ATTR_ID_STARTUP_ONOFF,
    #endif
} teCLD_OnOff_ClusterID;
```

14.9.2 teCLD_OOSC_SwitchType (On/Off Switch Types)

```
typedef enum
{
    E_CLD_OOSC_TYPE_TOGGLE,
    E_CLD_OOSC_TYPE_MOMENTARY
} teCLD_OOSC_SwitchType;
```

14.9.3 teCLD_OOSC_SwitchAction (On/Off Switch Actions)

```
typedef enum
{
    E_CLD_OOSC_ACTION_S2ON_S1OFF,
    E_CLD_OOSC_ACTION_S2OFF_S1ON,
    E_CLD_OOSC_ACTION_TOGGLE
} teCLD_OOSC_SwitchAction;
```

14.10 Compile-time options

- To enable the On/Off cluster in the code to be built, it is necessary to add the following to the **zcl_options.h** file:

```
#define CLD_ONOFF
```

- In addition, to include the software for a cluster client or server or both, it is necessary to add one or both of the following to the same file:

```
#define ONOFF_CLIENT
#define ONOFF_SERVER
```

- The On/Off cluster contains macros that may be optionally specified at compile time by adding some or all of the following lines to the **zcl_options.h** file.

Optional Attributes

1. To enable the optional On Configurable Duration attribute, add this line:

```
#define CLD_ONOFF_ATTR_ID_ON_CONFIGURABLE_DURATION
```

2. To enable the optional Duration Unit of Measure attribute, add this line:

```
#define CLD_ONOFF_ATTR_ID_DURATION_UNIT_OF_MEASUREMENT
```

3. To enable the optional Maximum Duration attribute, add this line:

```
#define CLD_ONOFF_ATTR_ID_MAX_DURATION
```

4. To enable the optional Global Scene Control attribute, add this line:

```
#define CLD_ONOFF_ATTR_GLOBAL_SCENE_CONTROL
```

5. To enable the optional On Time attribute, add this line:

```
#define CLD_ONOFF_ATTR_ON_TIME
```

6. To enable the optional Off Wait Time attribute, add this line:

```
#define CLD_ONOFF_ATTR_OFF_WAIT_TIME
```

7. To enable the optional Start-up On/Off attribute, add this line:

```
#define CLD_ONOFF_ATTR_STARTUP_ONOFF
```

8. To enable the optional Attribute Reporting Status attribute, add this line:

```
#define CLD_ONOFF_ATTR_ID_ATTRIBUTE_REPORTING_STATUS
```

Optional Commands

1. Add this line to enable processing of the On With Recall Global Scene command on the server:

```
#define CLD_ONOFF_CMD_ON_WITH_RECALL_GLOBAL_SCENE
```

2. Add this line to enable the optional On With Timed Off command on the client and server:

```
#define CLD_ONOFF_CMD_ON_WITH_TIMED_OFF
```

3. Add this line to enable the optional Off With Effect command on the client and server:

```
#define CLD_ONOFF_CMD_OFF_WITH_EFFECT
```

Cluster Revision

To define the value (n) of the Cluster Revision attribute, add this line:

```
#define CLD_ONOFF_CLUSTER_REVISION <n>
```

The default value is 1, which corresponds to the revision of the cluster in the ZCL r6 specification (see Section 2.4).

15 On/Off Switch Configuration Cluster

This chapter describes the On/Off Switch Configuration cluster.

The On/Off Switch Configuration cluster has a Cluster ID of 0x0007.

Note: *When using this cluster, the On/Off cluster must also be used (see Chapter 14).*

15.1 Overview

The On/Off Switch Configuration cluster allows the switch type on a device to be defined, as well as the commands to be generated when the switch is moved between its two states.

To use the functionality of this cluster, you must include the file **OOSC.h** in your application and enable the cluster by defining CLD_OOSC in the **zcl_options.h** file.

It is also necessary to enable the cluster as a server or client, or as both:

- The cluster server is able to receive commands to define a switch configuration.
- The cluster client is able to send commands to define a switch configuration.

The inclusion of the client or server software must be pre-defined in the compile-time options of the application. In addition, if the cluster is to reside on a custom endpoint then the role of client or server must also be specified when creating the cluster instance.

The compile-time options for the On/Off Switch Configuration cluster are fully detailed in Section 15.6.

15.2 On/Off Switch Config Cluster Structure and Attribute

The structure definition for the On/Off Switch Configuration cluster is:

```
typedef struct
{
#ifdef OOSC_SERVER
    zenum8                  eSwitchType;
    zenum8                  eSwitchActions;
#endif
    zuint16                 u16ClusterRevision;
} tsCLD_OOSC;
```

where:

- `eSwitchType` is the type of the switch, one of:
  - Toggle (0x00) - when the switch is physically moved between its two states, it remains in the latest state until it is physically returned to the original state (for example, a rocker switch)
  - Momentary (0x01) - when the switch is physically moved between its two states, it returns to the original state as soon as it is released (for example, a pushbutton which is pressed and then released)
  - Multi-function (0x02) - when the switch is physically moved between its two states, the command it sends is application-specific and may be dependent on the circumstances.

    Enumerations are provided for the above settings (see Section 15.5.2).
- `eSwitchActions` defines the commands to be generated when the switch moves between state 1 (S1) and state 2 (S2), one of:
  - S1 to S2 is ‘switch on’, S2 to S1 is ‘switch off’
  - S1 to S2 is ‘switch off’, S2 to S1 is ‘switch on’
  - S1 to S2 is ‘toggle’, S2 to S1 is ‘toggle’

    Enumerations are provided for the above settings (see Section 15.5.3).

- `u16ClusterRevision` is a mandatory attribute that specifies the revision of the cluster specification on which this cluster instance is based. The cluster specification in the ZCL r6 corresponds to a cluster revision of 1. The value is incremented by one for each subsequent revision of the cluster specification. This attribute is also described in Section2.4.

15.3 Initialisation

The function **eCLD_OOSCCreateOnOffSwitchConfig()** is used to create an instance of the On/Off Switch Configuration cluster. The function is called by the initialization function for the host device.

15.4 Functions

The following On/Off Switch Configuration cluster function is provided in the NXP implementation of the ZCL:

- eCLD_OOSCCreateOnOffSwitchConfig

15.4.1 eCLD_OOSCCreateOnOffSwitchConfig

```
teZCL_Status eCLD_OOSCCreateOnOffSwitchConfig(
         tsZCL_ClusterInstance *psClusterInstance,
         bool_t bIsServer,
         tsZCL_ClusterDefinition *psClusterDefinition,
         void *pvEndPointSharedStructPtr,
         tsZCL_AttributeStatus *psAttributeStatus);
```

Description

This function creates an instance of the On/Off Switch Configuration 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.

Call the function 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 creates an On/Off Switch Configuration cluster instance on the endpoint, but instances of other clusters may also be created on the same endpoint by calling their corresponding creation functions.

Note: *This function must not be called for an endpoint on which a standard ZigBee device is used. In this case, the device and its supported clusters must be registered on the endpoint using the relevant device registration function.*

When used, this function must be the first On/Off Switch Configuration cluster function called in the application, and must be called after the stack has been started and after the ZCL has been initialized.

Parameters

- *psClusterInstance* Pointer to structure containing information about the cluster instance to be created (see Section 6.1.16). This structure is updated by the function by initializing individual structure fields.
- *bIsServer* Type of cluster instance (server or client) to be created:
  TRUE - server
  FALSE - client
- *psClusterDefinition* Pointer to structure indicating the type of cluster to be created (see Section 6.1.2). In this case, this structure must contain the details of the On/Off Switch Configuration cluster. This parameter can refer to a pre-filled structure called `sCLD_OOSC` which is provided in the **OOSC.h** file.

- *pvEndPointSharedStructPtr* Pointer to the shared structure used for attribute storage. This parameter should be the address of the structure of type `tsCLD_OOSC` which defines the attributes of On/Off Switch Configuration cluster. The function initializes the attributes with default values.
- *psAttributeStatus* Pointer to a structure containing the storage for each attribute's status

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL

15.5 Enumerations

15.5.1 teCLD_OOSC_ClusterID

The following structure contains the enumerations used to identify the attributes of the On/Off Switch Configuration cluster.

```
typedef enum
{
    E_CLD_OOSC_ATTR_ID_SWITCH_TYPE        = 0x0000,   /* Mandatory */
    E_CLD_OOSC_ATTR_ID_SWITCH_ACTIONS     = 0x0010,   /* Mandatory */
} teCLD_OOSC_ClusterID;
```

15.5.2 teCLD_OOSC_SwitchType

The following structure contains the enumerations used to specify the switch type in the `eSwitchType` attribute.

```
typedef enum
{
    E_CLD_OOSC_TYPE_TOGGLE,
    E_CLD_OOSC_TYPE_MOMENTARY,
    E_CLD_OOSC_TYPE_MULTI_FUNCTION
} teCLD_OOSC_SwitchType;
```

The above enumerations are detailed in the table below.

Table 33. Switch Type Enumerations

Enumeration	Description
E_CLD_OOSC_TYPE_TOGGLE	Toggle - when the switch is physically moved between its two states, it remains in the latest state until it is physically returned to the original state (for example, a rocker switch).
E_CLD_OOSC_TYPE_MOMENTARY	Momentary - when the switch is physically moved between its two states, it returns to the original state as soon as it is released (for example, a pushbutton which is pressed and then released).
E_CLD_OOSC_TYPE_MULTI_FUNCTION	Multi-function - when the switch is physically moved between its two states, the command it sends is application-specific and may be dependent on the circumstances.

15.5.3 teCLD_OOSC_SwitchAction

The following structure contains the enumerations used to specify the switch action in the `eSwitchActions` attribute.

```
typedef enum
{
    E_CLD_OOSC_ACTION_S2ON_S1OFF,
    E_CLD_OOSC_ACTION_S2OFF_S1ON,
    E_CLD_OOSC_ACTION_TOGGLE
} teCLD_OOSC_SwitchAction;
```

The above enumerations are detailed in the table below.

Table 34. Switch Action Enumerations

Enumeration	Description When the switch moves between state 1 (S1) and state 2 (S2)...
E_CLD_OOSC_ACTION_S2ON_S1OFF	S1 to S2 is ‘switch on’, S2 to S1 is ‘switch off’
E_CLD_OOSC_ACTION_S2OFF_S1ON	S1 to S2 is ‘switch off’, S2 to S1 is ‘switch on’
E_CLD_OOSC_ACTION_TOGGLE	S1 to S2 is ‘toggle’, S2 to S1 is ‘toggle’

15.6 Compile-time options

To enable the On/Off Switch Configuration cluster in the code to be built, it is necessary to add the following to the **zcl_options.h** file:

```
#define CLD_OOSC
```

In addition, to include the software for a cluster client or server or both, it is necessary to add one or both of the following to the same file:

```
#define OOSC_CLIENT
#define OOSC_SERVER
```

To define the value (n) of the Cluster Revision attribute, add this line:

```
#define CLD_OOSC_CLUSTER_REVISION <n>
```

The default value is 1, which corresponds to the revision of the cluster in the ZCL r6 specification (see Section 2.4).

16 Level Control Cluster

This chapter describes the Level Control cluster.

The Level Control cluster has a Cluster ID of 0x0008.

16.1 Overview

The Level Control cluster is used to control the level of a physical quantity on a device. The physical quantity is device-dependent - for example, it could be light, sound or heat output.

Note: *This cluster should be used with the On/Off cluster (see Chapter 14) and this is assumed to be the case in this description.*

The Level Control cluster provides the facility to increase to a target level gradually during a 'switch-on' and decrease from this level gradually during a 'switch-off'.

To use the functionality of this cluster, you must include the file **LevelControl.h** in your application and enable the cluster by defining CLD_LEVEL_CONTROL in the **zcl_options.h** file.

It is also necessary to enable the cluster as a server or client, or as both:

- The cluster server is able to receive commands to change the level on the local device.
- The cluster client is able to send commands to change the level on the remote device.

The inclusion of the client or server software must be pre-defined in the application's compile-time options. In addition, if the cluster is to reside on a custom endpoint then the role of client or server must also be specified when creating the cluster instance.

The compile-time options for the Level Control cluster are fully detailed in Section 16.10.

16.2 Level Control Cluster structure and attributes

The structure definition for the Level Control cluster is shown below.

```
typedef struct
{
#ifdef LEVEL_CONTROL_SERVER
    zuint8                  u8CurrentLevel;
#ifdef CLD_LEVELCONTROL_ATTR_REMAINING_TIME
    zuint16                 u16RemainingTime;
#endif
    zbmap8                  u8Options;
#ifdef CLD_LEVELCONTROL_ATTR_ON_OFF_TRANSITION_TIME
    zuint16                 u16OnOffTransitionTime;
#endif
#ifdef CLD_LEVELCONTROL_ATTR_ON_LEVEL
    zuint8                  u8OnLevel;
#endif
#ifdef CLD_LEVELCONTROL_ATTR_ON_TRANSITION_TIME
    zuint16                 u16OnTransitionTime;
#endif
#ifdef CLD_LEVELCONTROL_ATTR_OFF_TRANSITION_TIME
    zuint16                 u16OffTransitionTime;
#endif
#ifdef CLD_LEVELCONTROL_ATTR_DEFAULT_MOVE_RATE
    zuint8                  u8DefaultMoveRate;
#endif
#ifdef CLD_LEVELCONTROL_ATTR_STARTUP_CURRENT_LEVEL
```

```
    zuint8                  u8StartUpCurrentLevel;
#endif
#ifdef CLD_LEVELCONTROL_ATTR_ATTRIBUTE_REPORTING_STATUS
    zenum8                  u8AttributeReportingStatus;
#endif
#endif
    zuint16                 u16ClusterRevision;
} tsCLD_LevelControl;
```

where:

- `u8CurrentLevel` is the current level on the device, in the range 0x01 to 0xFE (0x00 is not used and 0xFF represents an undefined level).
- `u16RemainingTime` is the time remaining (in tenths of a second) at the current level
- `u8Options` is a bitmap which allows behaviors connected with certain commands to be defined (these behaviors should only be defined during commissioning), as follows:

Bits	Name	Description
0	ChangeIfOff	Defines whether changes to the Level Control cluster can be made from control clusters (for example, Colour Control) when the `bOnOff` attribute of the On/Off cluster is zero (off): • 1 – Allow changes • 0 – Do not allow changes
1	CoupleColorTempToLevel	Defines whether changes to the `u8CurrentLevel` attribute are to be coupled with colour temperature: • 1 – Couple changes • 0 – Do not couple changes
2-7	-	Reserved

- `u16OnOffTransitionTime` is the time taken (in tenths of a second) to increase from ‘off’ to the target level or decrease from the target level to ‘off’ when an On or Off command is received, respectively (see below for target level)
- `u8OnLevel` is the target level to which `u8CurrentLevel` is set when an On command is received. The value must be in the range 0x01 to 0xFE. If maximum and minimum levels are implemented using the final four attributes of the cluster (see below), the value must be within the permissible range.
- `u16OnTransitionTime` is an optional attribute representing the time taken (in tenths of a second) to increase the level from 0 (off) to 255 (on) when an ‘On’ command of the On/Off cluster is received. The special value of 0xFFFF indicates that the transition time `u16OnOffTransitionTime` must be used instead (which is also used if `u16OnTransitionTime` is not enabled).
- `u16OffTransitionTime` is an optional attribute representing the time taken (in tenths of a second) to decrease the level from 255 (on) to 0 (off) when an ‘Off’ command of the On/Off cluster is received. The special value of 0xFFFF indicates that the transition time `u16OnOffTransitionTime` must be used instead (which is also be used if `u16OffTransitionTime` is not enabled).
- `u8DefaultMoveRate` is an optional attribute representing the rate of movement (in units per second) to be used when a Move command is received with a rate value (`u8Rate`) equal to 0xFF (see Section 16.8.3.2).
- `u8StartUpCurrentLevel` is an optional attribute that is used in the lighting domain to define the required start-up level of a light device when it is supplied with power. It determines the initial value of `u8CurrentLevel` on start-up (in the range 0x01 to 0xFE).
- `u8AttributeReportingStatus` is an optional attribute that should be enabled when attribute reporting is used for the cluster (see Section 2.3.5). The value of this attribute indicates whether there are attribute reports still pending (0x00) or the attribute reports are complete (0x01) - all other values are reserved. This attribute is also described in Section 2.4.

- `u16ClusterRevision` is a mandatory attribute that specifies the revision of the cluster specification on which this cluster instance is based. The cluster specification in the ZCL r6 corresponds to a cluster revision of 1. The value is incremented by one for each subsequent revision of the cluster specification. This attribute is also described in Section 2.4.

16.3 Attributes for Default Reporting

The following attribute of the Level Control cluster can be selected for default reporting:

- `u8CurrentLevel`

Attribute reporting (including default reporting) is described in Appendix B. Enabling reports for this attribute is described in Appendix B.3.6.

16.4 Initialization

The function **eCLD_LevelControlCreateLevelControl()** is used to create an instance of the Level Control cluster. The function is called by the initialization function for the host device.

16.5 Sending Remote Commands

The NXP implementation of the ZCL provides functions for sending commands between a Level Control cluster client and server. A command is sent from the client to one or more endpoints on the server. Multiple endpoints can usually be targeted using binding or group addressing.

16.5.1 Changing Level

Three functions (see below) are provided for sending commands to change the current level on a device. These commands modify the ‘current level’ attribute of the Level Control cluster.

Each of the three level functions can be implemented with the On/Off cluster. In this case:

- If the command increases the current level, the OnOff attribute of the On/Off cluster is set to ‘on’.
- If the command decreases the current level to the minimum permissible level for the device, the OnOff attribute of the On/Off cluster is set to ‘off’.

Use of the three functions/commands are described below.

Move to Level Command

The current level can be moved (up or down) to a new level over a given time using the function **eCLD_LevelControlCommandMoveToLevelCommandSend()**. The target level and transition time are specified in the command payload (see Section 16.8.3.1).

Move Command

The current level can be moved (up or down) at a specified rate using the function **eCLD_LevelControlCommandMoveCommandSend()**. The level varies until stopped (see Section 16.5.2) or until the maximum or minimum level is reached. The direction and rate are specified in the command payload (see Section 16.8.3.2).

Step Command

The current level can be moved (up or down) to a new level in a single step over a given time using the function **eCLD_LevelControlCommandStepCommandSend()**. The direction, step size and, transition time are specified in the command payload (see Section 16.8.3.3).

16.5.2 Stopping a Level Change

A level change initiated using any of the functions referenced in Section 16.5.1 can be halted using the function **eCLD_LevelControlCommandStopCommandSend()** or **eCLD_LevelControlCommandStopWithOnOff CommandSend()**.

16.6 Issuing Local Commands

Some of the operations described in Section 16.5 that correspond to remote commands can also be performed locally, as described below.

16.6.1 Setting Level

The level on the device on a local endpoint can be set using the function **eCLD_LevelControlSetLevel()**. This function sets the value of the 'current level' attribute of the Level Control cluster. A transition time must also be specified, in units of tenths of a second, during which the level moves toward the target value (this transition should be as smooth as possible, not stepped).

The specified level must be in the range 0x01 to 0xFE (the extreme values 0x00 and 0xFF are not used), where:

- 0x01 represents the minimum possible level for the device
- 0x02 to 0xFD are device-dependent values
- 0xFE represents the maximum level for the device

When the On/Off cluster is also enabled, calling the above function can have the following outcomes:

- If the operation is to increase the current level, the OnOff attribute of the On/Off cluster is set to 'on'.
- If the operation is to decrease the current level to the minimum permissible level for the device, the OnOff attribute of the On/Off cluster is set to 'off'.

16.6.2 Obtaining Level

The current level on the device on a local endpoint can be obtained using the function **eCLD_LevelControlGetLevel()**. This function reads the value of the 'current level' attribute of the Level Control cluster.

16.7 Functions

The following Level Control cluster functions are provided in the NXP implementation of the ZCL:

1. eCLD_LevelControlCreateLevelControl
2. eCLD_LevelControlSetLevel
3. eCLD_LevelControlGetLevel
4. eCLD_LevelControlCommandMoveToLevelCommandSend
5. eCLD_LevelControlCommandMoveCommandSend
6. eCLD_LevelControlCommandStepCommandSend
7. eCLD_LevelControlCommandStopCommandSend

16.7.1 eCLD_LevelControlCreateLevelControl

```
teZCL_Status eCLD_LevelControlCreateLevelControl(
 tsZCL_ClusterInstance *psClusterInstance,
 bool_t bIsServer,
```

```
 tsZCL_ClusterDefinition *psClusterDefinition,
 void *pvEndPointSharedStructPtr,
uint8 *pu8AttributeControlBits,
 tsCLD_LevelControlCustomDataStructure
 *psCustomDataStructure);
```

Description

This function creates an instance of the Level Control 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.

Call the function 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 creates a Level Control cluster instance on the endpoint, but instances of other clusters may also be created on the same endpoint by calling their corresponding creation functions.

Note: *This function must not be called for an endpoint on which a standard ZigBee device is used. In this case, the device and its supported clusters must be registered on the endpoint using the relevant device registration function.*

When used, this function must be the first Level Control cluster function called in the application, and must be called after the stack has been started and after the ZCL has been initialized.

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 Level Control cluster. The function initializes the array elements to zero.

Parameters

- *psClusterInstance*: Pointer to structure containing information about the cluster instance to be created (see Section 6.1.16). This structure is updated by the function by initializing individual structure fields.
- *bIsServer* Type of cluster instance (server or client) to be created:
  - TRUE - server
  - FALSE - client
- *psClusterDefinition*: Pointer to structure indicating the type of cluster to be created (see Section 6.1.2). In this case, this structure must contain the details of the Level Control cluster. This parameter can refer to a pre-filled structure called `sCLD_LevelControl` which is provided in the **LevelControl.h** file.
- *pvEndPointSharedStructPtr*: Pointer to the shared structure used for attribute storage. This parameter should be the address of the structure of type `tsCLD_LevelControl`, which defines the attributes of Level Control cluster. The function initializes the attributes with default values.
- *pu8AttributeControlBits*: Pointer to an array of **uint8** values, with one element for each attribute in the cluster (see above)
- *psCustomDataStructure*: Pointer to a structure containing the storage for internal functions of the cluster (see Section 16.8.2)

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL

16.7.2 eCLD_LevelControlSetLevel

```
teZCL_Status eCLD_LevelControlSetLevel(
```

```
  uint8 u8SourceEndPointId,
  uint8 u8Level,
  uint16 u16TransitionTime);
```

Description

This function sets the level on the device on the specified (local) endpoint by writing the specified value to the 'current level' attribute. The new level is implemented over the specified transition time by gradually changing the level.

The specified target level must be within the range 0x01 to 0xFE or a more restricted range imposed by the device manufacturer and/or user/installer (see Section 16.6.1).

This operation is performed with the On/Off cluster (if enabled), in which case:

- If the operation is to increase the current level, the OnOff attribute of the On/Off cluster is set to 'on'.
- If the operation is to decrease the current level to the minimum permissible level for the device, the OnOff attribute of the On/Off cluster is set to 'off'.

Parameters

- *u8SourceEndPointId*: Number of the local endpoint on which level is to be changed
- *u8Level*: New level to be set, within the range 0x01 to 0xFE or within a more restricted range (see above)
- *u16TransitionTime*: Time to be taken, in units of tenths of a second, to reach the target level (0xFFFF means move to the level as fast as possible)

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

16.7.3 eCLD_LevelControlGetLevel

```
teZCL_Status eCLD_LevelControlGetLevel(
  uint8 u8SourceEndPointId,
  uint8 *pu8Level);
```

Description

This function obtains the current level on the device on the specified (local) endpoint by reading the 'current level' attribute.

Parameters

- *u8SourceEndPointId* Number of the local endpoint from which the level is to be read

- *pu8Level* Pointer to location to receive obtained level

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

16.7.4 eCLD_LevelControlCommandMoveToLevelCommandSend

```
teZCL_Status eCLD_LevelControlCommandMoveToLevelCommandSend(
 uint8 u8SourceEndPointId,
 uint8 u8DestinationEndPointId,
 tsZCL_Address *psDestinationAddress,
 uint8 *pu8TransactionSequenceNumber,
 bool_t bWithOnOff,
 tsCLD_LevelControl_MoveToLevelCommandPayload
 *psPayload);
```

Description

This function sends a Move to Level command to instruct a device to move its ‘current level’ attribute to the specified level over a specified time. The new level and the transition time are specified in the payload of the command (see Section 16.8.3). The target level must be within the range 0x01 to 0xFE or a more restricted range imposed by the device manufacturer and/or user/installer (see Section 16.5.1).

The device receiving this message generates a callback event on the endpoint on which the Level Control cluster is registered and transition the ‘current level’ attribute to the new value.

The option is provided to use this command in association with the On/Off cluster. In this case:

- If the command is to increase the current level, the OnOff attribute of the On/Off cluster is set to ‘on’.
- If the command is to decrease the current level to the minimum permissible level for the device, the OnOff attribute of the On/Off cluster is set to ‘off’.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is 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 to send the request. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values
- *u8DestinationEndPointId*: Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP
- *psDestinationAddress*: Pointer to a structure holding the address of the node to which the request is sent

- *pu8TransactionSequenceNumber*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the request
- *bWithOnOff*: Specifies whether this cluster interacts with the On/Off cluster:
  - TRUE - interaction
  - FALSE - no interaction
- *psPayload*Pointer to a structure containing the payload for this message (see Section 16.8.3)

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

16.7.5 eCLD_LevelControlCommandMoveCommandSend

```
teZCL_Status eCLD_LevelControlCommandMoveCommandSend(
 uint8 u8SourceEndPointId,
 uint8 u8DestinationEndPointId,
 tsZCL_Address *psDestinationAddress,
 uint8 *pu8TransactionSequenceNumber,
 bool_t bWithOnOff,
tsCLD_LevelControl_MoveCommandPayload
         *psPayload);
```

Description

This function sends a Move command to instruct a device to move its ‘current level’ attribute either up or down in a continuous manner at a specified rate. The direction and rate are specified in the payload of the command (see Section 16.8.3).

If the current level reaches the maximum or minimum permissible level for the device, the level change stops.

The device receiving this message generates a callback event on the endpoint on which the Level Control cluster is registered, and move the current level in the direction and at the rate specified.

The option is provided to use this command in association with the On/Off cluster. In this case:

- If the command is to increase the current level, the OnOff attribute of the On/Off cluster is set to ‘on’.
- If the command decreases the current level to the minimum permissible level for the device, the OnOff attribute of the On/Off cluster is set to ‘off’.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is 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 to send the request. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values
- *u8DestinationEndPointId*: Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP
- *psDestinationAddress*: Pointer to a structure holding the address of the node to which the request is sent
- *pu8TransactionSequenceNumber*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the request
- *bWithOnOff*: Specifies whether this cluster interacts with the On/Off cluster:
  - TRUE - interaction
  - FALSE - no interaction
- *psPayload*: Pointer to a structure containing the payload for this message (see Section 16.8.3)

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

16.7.6 eCLD_LevelControlCommandStepCommandSend

```
teZCL_Status eCLD_LevelControlCommandStepCommandSend(
 uint8 u8SourceEndPointId,
 uint8 u8DestinationEndPointId,
 tsZCL_Address *psDestinationAddress,
 uint8 *pu8TransactionSequenceNumber,
 bool_t bWithOnOff,
tsCLD_LevelControl_StepCommandPayload
 *psPayload);
```

Description

This function sends a Step command to instruct a device to move its ‘current level’ attribute either up or down in a step of the specified step size over the specified time. The direction, step size and, transition time are specified in the payload of the command (see Section 16.8.3).

If the target level is above the maximum or below the minimum permissible level for the device, the stepped change is limited to this level, and the transition time is cut short.

The device receiving this message generates a callback event on the endpoint on which the Level Control cluster is registered and move the current level according to the specified direction, step size and transition time.

The option is provided to use this command in association with the On/Off cluster. In this case:

- If the command is to increase the current level, the OnOff attribute of the On/Off cluster is set to ‘on’.
- If the command decreases the current level to the minimum permissible level for the device, the OnOff attribute of the On/Off cluster is set to ‘off’.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is 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 to send the request. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values
- *u8DestinationEndPointId*: Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP
- *psDestinationAddress*: Pointer to a structure holding the address of the node to which the request is sent
- *pu8TransactionSequenceNumber*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the request
- *bWithOnOff*: Specifies whether this cluster interacts with the On/Off cluster:
  - TRUE - interaction
  - FALSE - no interaction
- *psPayload*: Pointer to a structure containing the payload for this message (see Section 16.8.3)

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

16.7.7 eCLD_LevelControlCommandStopCommandSend

```
teZCL_Status eCLD_LevelControlCommandStopCommandSend(
        uint8 u8SourceEndPointId,
        uint8 u8DestinationEndPointId,
        tsZCL_Address *psDestinationAddress,
        uint8 *pu8TransactionSequenceNumber,
        bool_t bWithOnOff,
        tsCLD_LevelControl_StopCommandPayload
 *psPayload);
```

Description

This function sends a Stop command to instruct a device to halt any transition to a new level. If necessary, the command can be sent as the ‘with On/Off’ version, used when the Level Control cluster interacts with the On/Off cluster, but the result on the target device is the same.

The device receiving this message generates a callback event on the endpoint on which the Level Control cluster is registered and stop any in progress transition.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is 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 to send the request. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values
- *u8DestinationEndPointId* Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP
- *psDestinationAddress* Pointer to a structure holding the address of the node to which the request is sent
- *pu8TransactionSequenceNumber* Pointer to a location to receive the Transaction Sequence Number (TSN) of the request
- *bWithOnOff* Specifies whether this cluster interacts with the On/Off cluster:
  TRUE - interaction
  FALSE - no interaction
- *psPayload* Pointer to a structure containing the payload for this message (see Section 16.8.3.4)

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

16.8 Structures

16.8.1 Level Control Transition Structure

The following structure is used to store information about an ongoing level transition.

```
typedef struct
{
    teCLD_LevelControl_Transition   eTransition;
    teCLD_LevelControl_MoveMode     eMode;
    bool bWithOnOff;
    int  iCurrentLevel;
    int  iTargetLevel;
    int  iPreviousLevel;
    int  iStepSize;
    uint32     u32Time;
} tsCLD_LevelControl_Transition;
```

where:

- `eTransition` is an enumeration indicating the type of level transition implemented - for the enumerations, see Section 16.9.2.
- `eMode` is an enumeration indicating the direction in which the level is moved during the transition - for the enumerations, see Section 16.9.3
- `bWithOnOff` is a boolean which is set to TRUE if the transition is implemented with the On/Off cluster (or FALSE otherwise). When enabled:
  - If the transition is to increase the level, the OnOff attribute of the On/Off cluster is set to ‘on’.
  - If the transition decreases the level to the minimum permissible level for the device, the OnOff attribute of the On/Off cluster is set to ‘off’.
- `iCurrentLevel` is the current level (0x01-0xFE) during the transition.
- `iTargetLevel` is the target level (0x01-0xFE) of the transition.
- `iPreviousLevel` is the previous level (0x01-0xFE) during the transition.
- `iStepSize` is the size of a single step of the transition.
- `u32Time` is the total time for the transition, in tenths of a second.

16.8.2 Custom Data Structure

The Level Control cluster requires extra storage space to be allocated for use by internal functions. The structure definition for this storage is shown below:

```
typedef struct
{
    tsCLD_LevelControl_Transition       sTransition;
    tsZCL_ReceiveEventAddress           sReceiveEventAddress;
    tsZCL_CallBackEvent                 sCustomCallBackEvent;
    tsCLD_LevelControlCallBackMessage   sCallBackMessage;
} tsCLD_LevelControlCustomDataStructure;
```

The fields are for internal use and no knowledge of them is required.

16.8.3 Custom Command Payloads

The following structures contain the payloads for the Level Control cluster custom commands.

16.8.3.1 Move To Level Command Payload

```
typedef struct
{
    uint8                   u8Level;
    uint16                  u16TransitionTime;
    zbmap8                  u8OptionsMask;
    zbmap8                  u8OptionsOverride;
} tsCLD_LevelControl_MoveToLevelCommandPayload;
```

where:

- `u8Level` is the target level within the range 0x01 to 0xFE or within a more restricted range (see Section 16.5.1)
- `u16TransitionTime` is the time taken, in units of tenths of a second, to reach the target level (0xFFFF means use the `u16OnOffTransitionTime` attribute instead - if this optional attribute is not present, the device changes the level as fast as possible).
- `OptionsMask` and `OptionsOverride` must be either both present or both not present. These fields are used in creating a temporary Options bitmap from the `u8Options` attribute. Each bit of the `u8Options`

attribute is carried across to the temporary Options bitmap unless the corresponding bit of `OptionsMask` is set (to 1). In this case, the corresponding bit of `OptionsOverride` is used in the temporary Options bitmap instead.

16.8.3.2 Move Command Payload

```
typedef struct
{
    uint8                   u8MoveMode;
    uint8                   u8Rate;
    zbmap8                  u8OptionsMask;
    zbmap8                  u8OptionsOverride;
} tsCLD_LevelControl_MoveCommandPayload;
```

where:

- `u8MoveMode` indicates the direction of the required level change, up (0x00) or down (0x01)
- `u8Rate` represents the required rate of change in units per second (0xFF means use the `u8DefaultMoveRate` attribute instead - if this optional attribute is not present, the device changes the level as fast as possible)
- `OptionsMask` and `OptionsOverride` must be either both present or both not present. These fields are used in creating a temporary Options bitmap from the `u8Options` attribute. Each bit of the `u8Options` attribute is carried across to the temporary Options bitmap unless the corresponding bit of `OptionsMask` is set (to 1). In this case, the corresponding bit of `OptionsOverride` is used in the temporary Options bitmap instead.

16.8.3.3 Step Command Payload

```
typedef struct
{
    uint8                   u8StepMode;
    uint8                   u8StepSize;
    uint16                  u16TransitionTime;
    zbmap8                  u8OptionsMask;
    zbmap8                  u8OptionsOverride;
} tsCLD_LevelControl_StepCommandPayload;
```

where:

- `u8StepMode` indicates the direction of the required level change, up (0x00) or down (0x01)
- `u8StepSize` is the size for the required level change
- `u16TransitionTime` is the time taken, in units of tenths of a second, to reach the target level (0xFFFF means move to the level as fast as possible)
- `OptionsMask` and `OptionsOverride` must be either both present or both not present. These fields are used in creating a temporary Options bitmap from the `u8Options` attribute. Each bit of the `u8Options` attribute is carried across to the temporary Options bitmap unless the corresponding bit of `OptionsMask` is set (to 1). In this case, the corresponding bit of `OptionsOverride` is used in the temporary Options bitmap instead.

16.8.3.4 Stop Command Payload

```
typedef struct
{
    zbmap8                  u8OptionsMask;
```

```
    zbmap8                  u8OptionsOverride;
} tsCLD_LevelControl_StopCommandPayload;
```

where `OptionsMask` and `OptionsOverride` must be either both present or both not present. These fields are used in creating a temporary Options bitmap from the `u8Options` attribute. Each bit of the `u8Options` attribute is carried across to the temporary Options bitmap unless the corresponding bit of `OptionsMask` is set (to 1). In this case, the corresponding bit of `OptionsOverride` is used in the temporary Options bitmap instead.

16.9 Enumerations

16.9.1 teCLD_LevelControl_ClusterID

The following enumerations are used to identify the attributes of the Level Control cluster.

```
typedef enum
{
    E_CLD_LEVELCONTROL_ATTR_ID_CURRENT_LEVEL                    = 0x0000,
    E_CLD_LEVELCONTROL_ATTR_ID_REMAINING_TIME,
    E_CLD_LEVELCONTROL_ATTR_ID_OPTIONS                          = 0x000F,
    E_CLD_LEVELCONTROL_ATTR_ID_ON_OFF_TRANSITION_TIME           = 0x010,
    E_CLD_LEVELCONTROL_ATTR_ID_ON_LEVEL,
    E_CLD_LEVELCONTROL_ATTR_ID_ON_TRANSITION_TIME,
    E_CLD_LEVELCONTROL_ATTR_ID_OFF_TRANSITION_TIME,
    E_CLD_LEVELCONTROL_ATTR_ID_DEFAULT_MOVE_RATE,
    E_CLD_LEVELCONTROL_ATTR_ID_STARTUP_CURRENT_LEVEL            = 0x4000,
} teCLD_LevelControl_ClusterID;
```

16.9.2 teCLD_LevelControl_Transition

```
The following enumerations are used to specify a type of level transition.
typedef enum
{
    E_CLD_LEVELCONTROL_TRANSITION_MOVE_TO_LEVEL = 0,
    E_CLD_LEVELCONTROL_TRANSITION_MOVE,
    E_CLD_LEVELCONTROL_TRANSITION_STEP,
    E_CLD_LEVELCONTROL_TRANSITION_STOP,
    E_CLD_LEVELCONTROL_TRANSITION_ON,
    E_CLD_LEVELCONTROL_TRANSITION_OFF,
    E_CLD_LEVELCONTROL_TRANSITION_OFF_WITH_EFFECT_DIM_DOWN_FADE_OFF,
    E_CLD_LEVELCONTROL_TRANSITION_OFF_WITH_EFFECT_DIM_UP_FADE_OFF,
    E_CLD_LEVELCONTROL_TRANSITION_NONE,
} teCLD_LevelControl_Transition;
```

16.9.3 teCLD_LevelControl_MoveMode

```
The following enumerations are used to specify the direction of a level change.
typedef enum
{
    E_CLD_LEVELCONTROL_MOVE_MODE_UP = 0x0,
    E_CLD_LEVELCONTROL_MOVE_MODE_DOWN
} teCLD_LevelControl_MoveMode;
```

16.10 Compile-time options

To enable the Level Control cluster in the code to be built, it is necessary to add the following to the **zcl_options.h** file:

```
#define CLD_LEVEL_CONTROL
```

In addition, to include the software for a cluster client or server or both, it is necessary to add one or both of the following to the same file:

```
#define LEVEL_CONTROL_CLIENT
#define LEVEL_CONTROL_SERVER
```

The Level Control cluster contains macros that may be optionally specified at compile time by adding one or more of the following lines to the **zcl_options.h** file.

Optional Attributes

To enable the optional Remaining Time attribute, add this line:

```
#define CLD_LEVELCONTROL_ATTR_REMAINING_TIME
```

To enable the optional On/Off Transition Time attribute, add this line:

```
#define CLD_LEVELCONTROL_ATTR_ON_OFF_TRANSITION_TIME
```

To enable the optional On Level attribute, add this line:

```
#define CLD_LEVELCONTROL_ATTR_ON_LEVEL
```

To enable the optional On Transition Time attribute, add this line:

```
#define CLD_LEVELCONTROL_ATTR_ON_TRANSITION_TIME
```

To enable the optional Off Transition Time attribute, add this line:

```
#define CLD_LEVELCONTROL_ATTR_OFF_TRANSITION_TIME
```

To enable the optional Default Move Rate attribute, add this line:

```
#define CLD_LEVELCONTROL_ATTR_DEFAULT_MOVE_RATE
```

To enable the optional Start-up Current Level attribute, add this line:

```
#define CLD_LEVELCONTROL_ATTR_STARTUP_CURRENT_LEVEL
```

Global Attributes

To enable the optional Attribute Reporting Status attribute, add this line:

```
#define CLD_LEVELCONTROL_ATTR_ID_ATTRIBUTE_REPORTING_STATUS
```

To define the value (n) of the Cluster Revision attribute, add this line:

```
#define CLD_LEVELCONTROL_CLUSTER_REVISION <n>
```

The default value is 1, which corresponds to the revision of the cluster in the ZCL r6 specification (see Section 2.4).

17 Alarms Cluster

This chapter describes the Alarms cluster which is defined in the ZCL.

The Alarms cluster has a Cluster ID of 0x0009.

17.1 Overview

The Alarms cluster is used to configure alarm functionality on a device and send alarm notifications to other devices.

Note: *The Alarms cluster is used with other clusters that use alarms. Alarms conditions and codes are cluster-specific and defined in these clusters.*

To use the functionality of this cluster, you must include the file **Alarms.h** in your application and enable the cluster by defining CLD_ALARMS in the **zcl_options.h** file.

An Alarms cluster instance can act as a client or a server. The inclusion of the client or server software must be pre-defined in the application's compile-time options (in addition, if the cluster is to reside on a custom endpoint then the role of client or server must also be specified when creating the cluster instance).

The compile-time options for the Alarms cluster are fully detailed in Section 17.9.

The Alarms cluster server resides on a device on which other clusters may generate alarm conditions (for example, a cluster attribute value exceeds a certain limit). When an alarm condition occurs, the Alarms cluster server may send an Alarm notification to a cluster client - for example, the client may be on a device that signals alarms to the user. An Alarms cluster client may also contain a user interface (for example, a set of buttons) which allows user instructions to be sent to the server - for example, to reset an alarm.

The Alarms cluster server implements alarm logging by keeping a record of the previously generated alarms in an Alarms table. Thus, historic alarm information can be retrieved from the Alarms table. Each entry of the table contains the following information about one alarm activation:

- Alarm code which identifies the type of alarm (this type is cluster-specific)
- Cluster ID of the cluster which generated the alarm
- Time-stamp indicating the time (UTC) at which the alarm is generated

A maximum number of Alarms table entries can be set in the compile-time options.

Note: *Any device which implements time-stamping for alarms must also employ the Time cluster, described in Chapter 17.*

17.2 Alarms Cluster structure and attributes

The structure definition for the Alarms cluster is shown below.

```
typedef struct
{
#ifdef ALARMS_SERVER
    #ifdef CLD_ALARMS_ATTR_ALARM_COUNT
        zuint16             u16AlarmCount;
    #endif
#endif
    zuint16                 u16ClusterRevision;
} tsCLD_Alarms;
```

where:

- `u16AlarmCount` is an optional attribute which contains the number of entries currently in the Alarms table on the cluster server.
- `u16ClusterRevision` is a mandatory attribute that specifies the revision of the cluster specification on which this cluster instance is based. The cluster specification in the ZCL r6 corresponds to a cluster revision of 1. The value is incremented by one for each subsequent revision of the cluster specification. This attribute is also described in Section 2.4.

17.3 Initialization

The function **eCLD_AlarmsCreateAlarms()** is used to create an instance of the Alarms cluster. The function is generally called by the initialization function for the host device.

17.4 Alarm Operations

This section describes the main operations that are performed using the Alarms cluster - raising an alarm and clearing/resetting an alarm.

17.4.1 Raising an Alarm

An alarm is raised when an alarm condition occurs on a cluster on the same endpoint as the Alarms cluster server - for example, when a cluster attribute falls below a lower threshold. The Alarms cluster server should then send an Alarm notification to any remote Alarms cluster clients that might be interested in the alarm. The server application can send this notification and add an entry to the Alarms table by calling the **eCLD_AlarmsSignalAlarm()** function. On arriving at a destination device, the notification causes an E_CLD_ALARMS_CMD_ALARM event to be generated to notify the client application.

17.4.2 Resetting Alarms (from Client)

A client application can remotely request one alarm or all alarms to be reset:

- The function **eCLD_AlarmsCommandResetAlarmCommandSend()** can be used to request an individual alarm to be reset. A Reset Alarm command is sent to the cluster server. On arriving at the destination device, the command causes an E_CLD_ALARMS_CMD_RESET_ALARM event to be generated.
- The function **eCLD_AlarmsCommandResetAllAlarmsCommandSend()** can be used to request all alarms to be reset. A Reset All Alarms command is sent to the cluster server. On arriving at the destination device, the command causes an E_CLD_ALARMS_CMD_RESET_ALL_ALARMS event to be generated.

On the generation of the above events on the cluster server, the server application can remove the relevant entry or entries from the local Alarms table as described in Section 17.4.2.

Note: The client application can also request that all the entries in an Alarms table are removed by calling **eCLD_AlarmsCommandResetAlarmLogCommandSend()**. In this case, the entries are automatically deleted by the ZCL on the server.

17.5 Alarms Events

The Alarms cluster has its own events that are handled through the callback mechanism outlined in Chapter 3. If a device uses the Alarms cluster, then Alarms 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 **eHA_RegisterThermostatEndPoint()** for a Thermostat device). The relevant callback function is then invoked when an Alarms event occurs.

For an Alarms 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 tsCLD_AlarmsCallBackMessage structure:

```
typedef struct
{
 uint8  u8CommandId;
 union
  {
 tsCLD_AlarmsResetAlarmCommandPayload  *psResetAlarmCommandPayload;
  tsCLD_AlarmsAlarmCommandPayload  *psAlarmCommandPayload;
  tsCLD_AlarmsGetAlarmResponsePayload   *psGetAlarmResponse;
  } uMessage;
} tsCLD_AlarmsCallBackMessage;
```

When an Alarms event occurs, one of a number of command types could have been received. The relevant command type is specified through the u8CommandId field of the tsCLD_AlarmsCallBackMessage structure. The possible command types are detailed below.

The table below lists and describes the command types that can be received by the cluster server.

Table 35. Alarms Command Types (on Server)

u8CommandId Enumeration	Description
E_CLD_ALARMS_CMD_RESET_ALARM	A Reset Alarm command has been received
E_CLD_ALARMS_CMD_RESET_ALL_ALARMS	A Reset All Alarms command has been received
E_CLD_ALARMS_CMD_GET_ALARM	A Get Alarm command has been received
E_CLD_ALARMS_CMD_RESET_ALARM_LOG	A Reset Alarm Log command has been received

The table below lists and describes the command types that can be received by the cluster client.

Table 36. Alarms Command Types (on Client)

u8CommandId Enumeration	Description
E_CLD_ALARMS_CMD_ALARM	An Alarm notification has been received
E_CLD_ALARMS_CMD_GET_ALARM_RESPONSE	A Get Alarm response has been received

17.6 Functions

The following Alarms cluster functions are provided in the NXP implementation of the ZCL:

1. eCLD_AlarmsCreateAlarms
2. eCLD_AlarmsCommandResetAlarmCommandSend
3. eCLD_AlarmsCommandResetAllAlarmsCommandSend
4. eCLD_AlarmsCommandGetAlarmCommandSend
5. eCLD_AlarmsCommandResetAlarmLogCommandSend
6. eCLD_AlarmsResetAlarmLog
7. eCLD_AlarmsAddAlarmToLog
8. eCLD_AlarmsGetAlarmFromLog

17.6.1 eCLD_AlarmsCreateAlarms

```
teZCL_Status eCLD_AlarmsCreateAlarms(
 tsZCL_ClusterInstance *psClusterInstance,
```

```
  bool_t bIsServer,
  tsZCL_ClusterDefinition *psClusterDefinition,
  void  *pvEndPointSharedStructPtr,
  uint8 *pu8AttributeControlBits,
  tsCLD_AlarmsCustomDataStructure
        *psCustomDataStructure);
```

Description

This function creates an instance of the Alarms 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 is only 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 creates an Alarms cluster instance on the endpoint, but instances of other clusters may also be created on the same endpoint by calling their corresponding creation functions.

Note: *This function is not called for an endpoint on which a standard ZigBee device is used. In this case, the device and its supported clusters must be registered on the endpoint using the relevant device registration function.*

When used, this function must be the first Alarms cluster function called in the application, and must be called after the stack has been started and after the ZCL has been initialized.

Parameters

- *psClusterInstance*:Pointer to structure containing information about the cluster instance to be created (see Section 6.1.16). This structure is updated by the function by initializing individual structure fields.
- *bIsServer*: Type of cluster instance (server or client) to be created:
  - TRUE - server
  - FALSE - client
- *psClusterDefinition*: Pointer to structure indicating the type of cluster to be created (see Section 6.1.2). In this case, this structure must contain the details of the Alarms cluster. This parameter can refer to a pre-filled structure called `sCLD_Alarms` which is provided in the **Alarms.h** file.
- *pvEndPointSharedStructPtr*: Pointer to the shared structure used for attribute storage. This parameter should be the address of the structure of type `tsCLD_Alarms` which defines the attributes of Alarms cluster. The function initializes the attributes with default values.
- *pu8AttributeControlBits*: Pointer to an array of **uint8** values, with one element for each attribute in the cluster (see above)
- *psCustomDataStructure*: Pointer to a structure containing the storage for internal functions of the cluster (see Section 17.7.2)

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL

17.6.2 eCLD_AlarmsCommandResetAlarmCommandSend

```
teZCL_Status eCLD_AlarmsCommandResetAlarmCommandSend(
 uint8 u8SourceEndPointId,
 uint8 u8DestinationEndPointId,
```

```
  tsZCL_Address *psDestinationAddress,
  uint8 *pu8TransactionSequenceNumber,
  tsCLD_AlarmsResetAlarmCommandPayload
       *psPayload);
```

Description

This function can be called on an Alarms cluster client to send a Reset Alarm command to a cluster server. This command requests that a specific alarm for a specific cluster is reset. The function may be called as the result of user input. The relevant alarm and cluster ID must be specified in the command payload (see Section 17.7.3.1).

On receiving the command, an E_CLD_ALARMS_CMD_RESET_ALARM event is generated on the cluster server to notify the application.

The function is only used to reset alarms that are not automatically reset when the alarm condition no longer exists.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is 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 to send the request. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values
- *u8DestinationEndPointId*: Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP
- *psDestinationAddress*: Pointer to a structure holding the address of the node to which the request is sent
- *pu8TransactionSequenceNumber*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the request
- *psPayload*: Pointer to a structure containing the payload for the command (see Section 17.7.3.1)

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

17.6.3 eCLD_AlarmsCommandResetAllAlarmsCommandSend

```
teZCL_Status eCLD_AlarmsCommandResetAllAlarmsCommandSend(
 uint8 u8SourceEndPointId,
 uint8 u8DestinationEndPointId,
 tsZCL_Address *psDestinationAddress,
```

```
  uint8 *pu8TransactionSequenceNumber);
```

Description

This function can be called on an Alarms cluster client to send a Reset All Alarms command to a cluster server. This command requests that all alarms on the server device are reset. The function may be called as the result of user input.

On receiving the command, an E_CLD_ALARMS_CMD_RESET_ALL_ALARMS event is generated on the cluster server to notify the application.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is 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 to send the request. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values
- *u8DestinationEndPointId*: Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP
- *psDestinationAddress*: Pointer to a structure holding the address of the node to which the request is sent
- *pu8TransactionSequenceNumber*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the request

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

17.6.4 eCLD_AlarmsCommandGetAlarmCommandSend

```
teZCL_Status eCLD_AlarmsCommandGetAlarmCommandSend(
 uint8 u8SourceEndPointId,
 uint8 u8DestinationEndPointId,
 tsZCL_Address *psDestinationAddress,
 uint8 *pu8TransactionSequenceNumber);
```

Description

This function can be used on an Alarms cluster client to send a Get Alarm command to a cluster server. This command requests information on the logged alarm with the earliest timestamp in the device's Alarms table. As a result of this command, the retrieved entry is also deleted from the table.

The requested information is returned by the server in a Get Alarm response. When this response is received, an E_CLD_ALARMS_CMD_GET_ALARM_RESPONSE event is generated on the client.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is 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 to send the request. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values
- *u8DestinationEndPointId*: Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP
- *psDestinationAddress*: Pointer to a structure holding the address of the node to which the request is sent
- *pu8TransactionSequenceNumber*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the request

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

17.6.5 eCLD_AlarmsCommandResetAlarmLogCommandSend

```
teZCL_Status eCLD_AlarmsCommandResetAlarmLogCommandSend(
  uint8 u8SourceEndPointId,
  uint8 u8DestinationEndPointId,
  tsZCL_Address *psDestinationAddress,
  uint8 *pu8TransactionSequenceNumber);
```

Description

This function can be called on an Alarms cluster client to send a Reset Alarm Log command to a cluster server. This command requests that the Alarms table on the server is cleared of all entries. The function may be called as the result of user input.

On receiving the command, an E_CLD_ALARMS_CMD_RESET_ALARM_LOG event is generated on the cluster server to notify the application but the ZCL automatically clears the Alarms table.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is 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 to send the request. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values
- *u8DestinationEndPointId*: Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP
- *psDestinationAddress*: Pointer to a structure holding the address of the node to which the request is sent
- *pu8TransactionSequenceNumber*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the request

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

17.6.6 eCLD_AlarmsResetAlarmLog

```
teZCL_Status eCLD_AlarmsResetAlarmLog(
 tsZCL_EndPointDefinition *psEndPointDefinition,
 tsZCL_ClusterInstance *psClusterInstance);
```

Description

This function can be called on the Alarms cluster server to clear all entries of the local Alarms table. The function may be called as the result of user input.

Parameters

- *psEndPointDefinition*: Pointer to the ZCL endpoint definition structure for the application (see Section 6.1.1)
- *psClusterInstance*: Pointer to structure containing information about the Alarms cluster instance (see Section 6.1.16)

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

17.6.7 eCLD_AlarmsAddAlarmToLog

```
teZCL_Status eCLD_AlarmsAddAlarmToLog(
 tsZCL_EndPointDefinition *psEndPointDefinition,
 tsZCL_ClusterInstance *psClusterInstance,
 uint8 u8AlarmCode,
 uint16 u16ClusterId);
```

Description

This function can be called on the Alarms cluster server to add a new entry to the local Alarms table. The function should be called by the server application when an alarm condition has occurred. The alarm and the cluster which generated it must be specified. A timestamp (UTC) for the alarm is automatically inserted into the entry.

Parameters

- *psEndPointDefinition*: Pointer to the ZCL endpoint definition structure for the application (see Section 6.1.1)
- *psClusterInstance*: Pointer to structure containing information about the Alarms cluster instance (see Section 6.1.16)
- *u8AlarmCode*: Code that identifies the type of alarm to be added
- *u16ClusterId*: Cluster ID of the cluster which generated the alarm

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

17.6.8 eCLD_AlarmsGetAlarmFromLog

```
teZCL_Status eCLD_AlarmsGetAlarmFromLog(
  tsZCL_EndPointDefinition *psEndPointDefinition,
  tsZCL_ClusterInstance *psClusterInstance,
  uint8 *pu8AlarmCode,
  uint16 *pu16ClusterId,
  uint32 *pu32TimeStamp);
```

Description

This function can be called on the Alarms cluster server to obtain an entry from the local Alarms table. Information on the logged alarm with the earliest timestamp in the device's Alarms table is returned - pointers to memory locations to receive the retrieved alarm data must be provided. As a result of this command, the retrieved entry is also deleted from the table.

Parameters

- *psEndPointDefinition*: Pointer to the ZCL endpoint definition structure for the application (see Section 6.1.1)
- *psClusterInstance*: Pointer to structure containing information about the Alarms cluster instance (see Section 6.1.16)
- *pu8AlarmCode*: Pointer to location to receive the alarm code which identifies the retrieved alarm type
- *pu16ClusterId*: Pointer to location to receive the Cluster ID of the cluster which generated the alarm
- *pu32TimeStamp*: Pointer to location to receive timestamp (UTC) of the retrieved alarm (a value of 0XFFFFFFFF indicates that no timestamp is available for the alarm)

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

17.6.9 eCLD_AlarmsSignalAlarm

```
teZCL_Status eCLD_AlarmsSignalAlarm(
    uint8 u8SourceEndPointId,
    uint8 u8DestinationEndPointId,
    tsZCL_Address *psDestinationAddress,
    uint8 *pu8TransactionSequenceNumber,
    uint8 u8AlarmCode,
    uint16 u16ClusterId);
```

Description

This function can be called on the Alarms cluster server to send an Alarm notification to a cluster client and add a log entry to the local Alarms table on the server. The function should be called by the server application when an alarm condition has occurred. The alarm and the cluster which generated it must be specified.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is 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 to send the request. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values
- *u8DestinationEndPointId*: Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP
- *psDestinationAddress*: Pointer to a structure holding the address of the node to which the request is sent
- *pu8TransactionSequenceNumber*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the request
- *u8AlarmCode*: Code which identifies the type of alarm that has occurred
- *u16ClusterId*: Cluster ID of the cluster which generated the alarm

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

17.7 Structures

17.7.1 Event Callback Message Structure

For an Alarms 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 `tsCLD_AlarmsCallBackMessage` structure:

```
typedef struct
{
   uint8  u8CommandId;
   union
   {
 tsCLD_AlarmsResetAlarmCommandPayload *psResetAlarmCommandPayload;
 tsCLD_AlarmsAlarmCommandPayload   *psAlarmCommandPayload;
 tsCLD_AlarmsGetAlarmResponsePayload  *psGetAlarmResponse;
} uMessage;
} tsCLD_AlarmsCallBackMessage;
```

where:

- `u8CommandId` indicates the type of Alarms command that has been received by a cluster server or client, one of:
  - E_CLD_ALARMS_CMD_RESET_ALARM (server event)
  - E_CLD_ALARMS_CMD_RESET_ALL_ALARMS (server event)
  - E_CLD_ALARMS_CMD_GET_ALARM (server event)

  - E_CLD_ALARMS_CMD_RESET_ALARM_LOG (server event)
  - E_CLD_ALARMS_CMD_ALARM (client event)
  - E_CLD_ALARMS_CMD_GET_ALARM_RESPONSE (client event)
- `uMessage` is a union containing the command payload in the following form:
  - `psResetAlarmCommandPayload` is a pointer to a structure containing the Reset Alarm command payload - see Section 17.7.3.1
  - `psAlarmCommandPayload` is a pointer to a structure containing the Alarm notification payload - see Section 17.7.3.2
  - `psGetAlarmResponse` is a pointer to a structure containing the Get Alarm response payload - see Section 17.7.4.1

For further information on the above events, refer to Section 17.5.

17.7.2 Custom Data Structure

The Alarms cluster requires extra storage space to be allocated for use by internal functions. The structure definition for this storage is shown below:

```
typedef struct
{
  DLIST   lAlarmsAllocList;
  DLIST   lAlarmsDeAllocList;
  tsZCL_ReceiveEventAddress       sReceiveEventAddress;
  tsZCL_CallBackEvent             sCustomCallBackEvent;
  tsCLD_AlarmsCallBackMessage     sCallBackMessage;
  tsCLD_AlarmsTableEntry
  asAlarmsTableEntry[CLD_ALARMS_MAX_NUMBER_OF_ALARMS];
} tsCLD_AlarmsCustomDataStructure;
```

The fields are for internal use and no knowledge of them is required.

17.7.3 Custom Command Payloads

This section contains the structures for the payloads of the Alarms cluster custom commands.

17.7.3.1 Reset Alarm Command Payload

```
typedef struct
{
   uint8   u8AlarmCode;
   uint16  u16ClusterId;
} tsCLD_AlarmsResetAlarmCommandPayload;
```

where:

- `u8AlarmCode` is the code which identifies the type of alarm to be reset - these codes are cluster-specific
- `u16ClusterId` is the Cluster ID of the cluster which generates the alarm to be reset

17.7.3.2 Alarm Notification Payload

```
typedef struct
{
  uint8   u8AlarmCode;
  uint16   u16ClusterId;
```

```
} tsCLD_AlarmsAlarmCommandPayload;
```

where:

- u8AlarmCode is the code which identifies the type of alarm that has been generated - these codes are cluster-specific
- u16ClusterId is the Cluster ID of the cluster which generated the alarm

17.7.4 Custom Response Payloads

This section contains the structures for the payloads of the Alarms cluster custom responses.

17.7.4.1 Get Alarm Response Payload

```
typedef struct
{
   uint8   u8Status;
   uint8   u8AlarmCode;
   uint16  u16ClusterId;
   uint32  u32TimeStamp;
} tsCLD_AlarmsGetAlarmResponsePayload;
```

where:

- u8Status indicates the result of the Get Alarm operation as follows:
  - SUCCESS (0x01): An alarm entry is successfully retrieved from the Alarms table and its details are reported in the remaining fields (below)
  - NOT_FOUND (0x00): There were no alarm entries to be retrieved from the Alarms table and the remaining fields (below) are empty
- u8AlarmCode is the code which identifies the type of alarm reported - these codes are cluster-specific
- u16ClusterId is the Cluster ID of the cluster which generated the alarm
- u32TimeStamp is a timestamp representing the time (UTC) at which the alarm was generated (a value of 0XFFFFFFFF indicates that no timestamp is available for the alarm)

17.7.5 Alarms Table Entry

The following structure contains the data for an entry of an Alarms table.

```
typedef struct
{
    DNODE    dllAlarmsNode;
    uint8    u8AlarmCode;
    uint16   u16ClusterId;
    uint32   u32TimeStamp;
} tsCLD_AlarmsTableEntry;
```

where:

- dllAlarmsNode is for internal use and no knowledge of it is required
- u8AlarmCode is the code which identifies the type of alarm - these codes are cluster-specific
- u16ClusterId is the Cluster ID of the cluster which generated the alarm
- u32TimeStamp is a timestamp representing the time (UTC) at which the alarm was generated (a value of 0XFFFFFFFF indicates that no timestamp is available for the alarm)

17.8 Enumerations

17.8.1 teCLD_Alarms_AttributeID

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

```
typedef enum
{
  E_CLD_ALARMS_ATTR_ID_ALARM_COUNT = 0x0000,
} teCLD_Alarms_AttributeID;
```

18 Time Cluster and ZCL Time

This chapter describes the Time cluster which is defined in the ZCL. This cluster is used to maintain a time reference for the transactions in a ZigBee PRO network and to time synchronize the ZigBee PRO devices.

The Time cluster has a Cluster ID of 0x000A.

This section also describes the maintenance of 'ZCL time'.

18.1 Overview

The Time cluster is required in a ZigBee PRO network in which the constituent devices must be kept time-synchronized - for example, in an HVAC system, it may be necessary for heating to operate only between specific times of the day. In such a case, one device implements the Time cluster as a server and acts as the time-master for the network. While the other devices in the network, implement the Time cluster as a client and time-synchronize with the server.

Note: *As for all clusters, the Time cluster is stored in a shared device structure (see Section 18.3) which, for the cluster client, reflects the state of the cluster server. Access to the shared device structure (on Time cluster server and client) must be controlled using a mutex - for information on mutexes, refer to Appendix A.*

The Time cluster is enabled by defining CLD_TIME in the **zcl_options.h** file. The inclusion of the client or server software must also be pre-defined in the application's compile-time options. In addition, if the cluster is to reside on a custom endpoint then specify the role of client or server, when creating the cluster instance. The compile-time options for the Time cluster are fully detailed in Section 18.10.

In addition to the time in the Time cluster, the ZCL also keeps its own time, 'ZCL time'. ZCL time may be maintained on a device even when the Time cluster is not used by the device. Both times are described below.

Time Attribute

The Time cluster contains an attribute for the current time, as well as associated information such as time-zone and daylight saving - see Section 18.3. The time attribute is referenced to UTC (Coordinated 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 January 1, 2000 UTC".

ZCL Time

'ZCL time' is based on the above **UTCTime** definition. This time is derived from a 1 second timer and is used to drive any ZCL timers that have been registered.

18.2 Time Cluster structure and attributes

The Time cluster is contained in the following `tsCLD_Time` structure:

```
typedef struct
{
#ifdef TIME_SERVER
    zutctime                utctTime;
    zbmap8                  u8TimeStatus;
#ifdef CLD_TIME_ATTR_TIME_ZONE
    zint32                  i32TimeZone;
#endif
#ifdef CLD_TIME_ATTR_DST_START
    zuint32                 u32DstStart;
```

```
#endif
#ifdef CLD_TIME_ATTR_DST_END
    zuint32                u32DstEnd;
#endif
#ifdef CLD_TIME_ATTR_DST_SHIFT
    zint32                 i32DstShift;
#endif
#ifdef CLD_TIME_ATTR_STANDARD_TIME
    zuint32                u32StandardTime;
#endif
#ifdef CLD_TIME_ATTR_LOCAL_TIME
    zuint32                u32LocalTime;
#endif
#ifdef CLD_TIME_ATTR_LAST_SET_TIME
    zutctime                u32LastSetTime;
#endif
#ifdef CLD_TIME_ATTR_VALID_UNTIL_TIME
    zutctime                u32ValidUntilTime;
#endif
#endif
    zuint16                 u16ClusterRevision;
} tsCLD_Time;;
```

where:

- `utctTime` is a mandatory 32-bit attribute which holds the current time (UTC). This attribute can only be over-written using a remote ‘write attributes’ request if the local Time cluster is not configured as the time-master for the network - this is the case if bit 0 of the element `u8TimeStatus` (see below) is set to 0.
- `u8TimeStatus` is a mandatory 8-bit attribute containing the following bitmap:

Table 37. u8TimeStatus Bitmap

Bits	Meaning	Description
0	Master	1: Time-master for network 0: Not time-master for network
1	Synchronized	1: Synchronized to another device 0: Not synchronized to another 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	-

* DST= Daylight Saving Time

Macros are provided for setting the individual bits of this bitmap:

– CLD_TM_TIME_STATUS_MASTER_MASK (bit 0)

– CLD_TM_TIME_STATUS_SYNCHRONIZED_MASK (bit 1)

– CLD_TM_TIME_STATUS_MASTER_ZONE_DST_MASK (bit 2)

- `i32TimeZone` is an optional attribute which indicates the local time-zone expressed as an offset from UTC, in seconds.
- `u32DstStart` is an optional attribute which contains the start-time (UTC), in seconds, for daylight saving for the current year.
- `u32DstEnd` is an optional attribute which contains the end-time (UTC), in seconds, for daylight saving for the current year.
- `i32DstShift` is an optional attribute which contains the local time-shift, in seconds, relative to standard local time that is applied during the daylight saving period.

- `u32StandardTime` is an optional attribute which contains the local standard time (equal to `utctTime` + `i32TimeZone`).
- `u32LocalTime` is an optional attribute which contains the local time taking into account daylight saving, if applicable (equal to `utctTime` + `i32TimeZone` + `i32DstShift` during the daylight saving period).
- `u32LastSetTime` is an optional attribute which indicates the most recent UTC time at which the Time attribute (`utctTime`) was set, either internally or over the ZigBee network.
- `u32ValidUntilTime` is an optional attribute which indicates a UTC time (later than `u32LastSetTime`) up to which the Time attribute (`utctTime`) value may be trusted.
- `u16ClusterRevision` is a mandatory attribute that specifies the revision of the cluster specification on which this cluster instance is based. The cluster specification in the ZCL r6 corresponds to a cluster revision of 1. The value is incremented by one for each subsequent revision of the cluster specification. This attribute is also described in Section 2.4.

Note: *If necessary, the daylight saving attributes (`u32DstStart`, `u32DstEnd` and, `i32DstShift`) must all be enabled together.*

The Time cluster structure contains two mandatory elements, `utctTime` and `u8TimeStatus`. The remaining elements are optional, each being enabled/disabled through a corresponding macro defined in the **zcl_options.h** file - for example, the optional time zone element `i32TimeZone` is enabled/disabled through the macro CLD_TIME_ATTR_TIME_ZONE (see Section 18.3.2).

18.3 Attribute Settings

18.3.1 Mandatory Attributes

The mandatory attributes of the Time cluster are set as follows:

utctTime

This is a mandatory 32-bit attribute which holds the current time (UTC). On the time-master, this attribute value is incremented once per second. On all other 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 18.5.

u8TimeStatus

This is a mandatory 8-bit attribute containing the bitmap detailed in Table 28 on page 389. This attribute must be set as follows on the time-master (Time cluster server):

- The ‘Master’ bit should initially be zero until the current time has been obtained from an 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 time-master does not obtain its time from another device within the ZigBee network (this bit is set to ‘1’ only for a secondary Time cluster server that is synchronized 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 `u8TimeStatus` bitmap - for example, the macro CLD_TM_TIME_STATUS_MASTER_MASK is used to set the Master bit. These macros are defined in the header file **time.h** and are also listed in Section 18.2.

18.3.2 Optional Attributes

The optional attributes of the Time cluster are set as follows:

i32TimeZone

This is an optional attribute which is enabled using the macro CLD_TIME_ATTR_TIME_ZONE and which indicates the local time-zone.

The local time-zone is expressed as an offset from UTC, where this offset is quantified in seconds. Therefore:

Current local standard time = `utctTime` + `i32TimeZone`

where `i32TimeZone` is negative if the local time is behind UTC.

u32DstStart

This is an optional attribute which is enabled using the macro CLD_TIME_ATTR_DST_START and which contains the start-time (in seconds) for daylight saving for the current year.

If `u32DstStart` is used then `u32DstEnd` and `i32DstShift` are also required.

u32DstEnd

This is an optional attribute which is enabled using the macro CLD_TIME_ATTR_DST_END and which contains the end-time (in seconds) for daylight saving for the current year.

If `u32DstEnd` is used then `u32DstStart` and `i32DstShift` are also required.

i32DstShift

This is an optional attribute which is enabled using the macro CLD_TIME_ATTR_DST_SHIFT and which contains the local time-shift (in seconds), relative to standard local time, that is applied during the daylight saving period (between `u32DstStart` and `u32DstEnd`). During this period:

Current local time = `utctTime` + `i32TimeZone` + `i32DstShift`

This time-shift varies between territories, but is 3600 seconds (1 hour) for Europe and North America.

If `i32DstShift` is used then `u32DstStart` and `u32DstEnd` are also required.

u32StandardTime

This is an optional attribute which is enabled using the macro CLD_TIME_ATTR_STANDARD_TIME and which contains the local standard time (equal to `utctTime` + `i32TimeZone`).

u32LocalTime

This is an optional attribute which is enabled using the macro CLD_TIME_ATTR_LOCAL_TIME and which contains the local time taking into account daylight saving, if applicable (equal to `utctTime` + `i32TimeZone` + `i32DstShift` during the daylight saving period and equal to `u32StandardTime` outside of the daylight saving period).

u32LastSetTime

This is an optional attribute which is enabled using the macro CLD_TIME_ATTR_LAST_SET_TIME and which indicates the most recent UTC time at which the Time attribute (`utctTime`) was set, either internally or over the ZigBee network.

u32ValidUntilTime

This is an optional attribute which is enabled using the macro CLD_TIME_ATTR_VALID_UNTIL_TIME and indicates a UTC time (later than `u32LastSetTime`) up to which the Time attribute (`utctTime`) value may be trusted.

18.4 Maintaining ZCL Time

The simplest case of keeping time on a ZigBee PRO device is to maintain 'ZCL time' only (without using the Time cluster). In this case, the ZCL time on a device can be initialized by the application using the function **vZCL_SetUTCTime()**.

The ZCL time is subsequently incremented from a local one-second timer, as follows. On expiration of the timer, an event is generated (from the hardware/software timer that drives the one-second timer), which causes a ZCL user task to be activated. The event is initially handled by this task as described in Section 3.2, 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.
2. The user task resumes the one-second timer.

18.4.1 Updating ZCL Time Following 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 device normally uses its RC oscillator for timing purposes, which may not maintain the required accuracy. 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. Note that when passed into **vZCL_EventHandler()**, this event will increment the ZCL time by one second.

18.4.2 ZCL Time Synchronization

The local ZCL time on a device can be synchronized with the time in a time-related cluster, such as Time, Price, or Messaging. The ZCL time is considered to be synchronized following a call to **vZCL_SetUTCTime()**. The NXP implementation of the ZCL also provides the following functions relating to ZCL time synchronization:

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

18.5 Time-Synchronization of Devices

The devices in a ZigBee PRO network may need to be time-synchronized (so that they all refer to the same time). In this case, the Time cluster is used and one device acts as the Time cluster server and time-master from which the other devices set their time.

Note: *Synchronization with a time-master is not required in all networks. In such cases, it is sufficient to use the ZCL time without synchronization between devices, as described in Section 18.4.*

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 initialized by the local application using an external master time and are subsequently maintained using a local one-second timer (see Section 18.5.1), as well as occasional re-synchronizations with external master time.

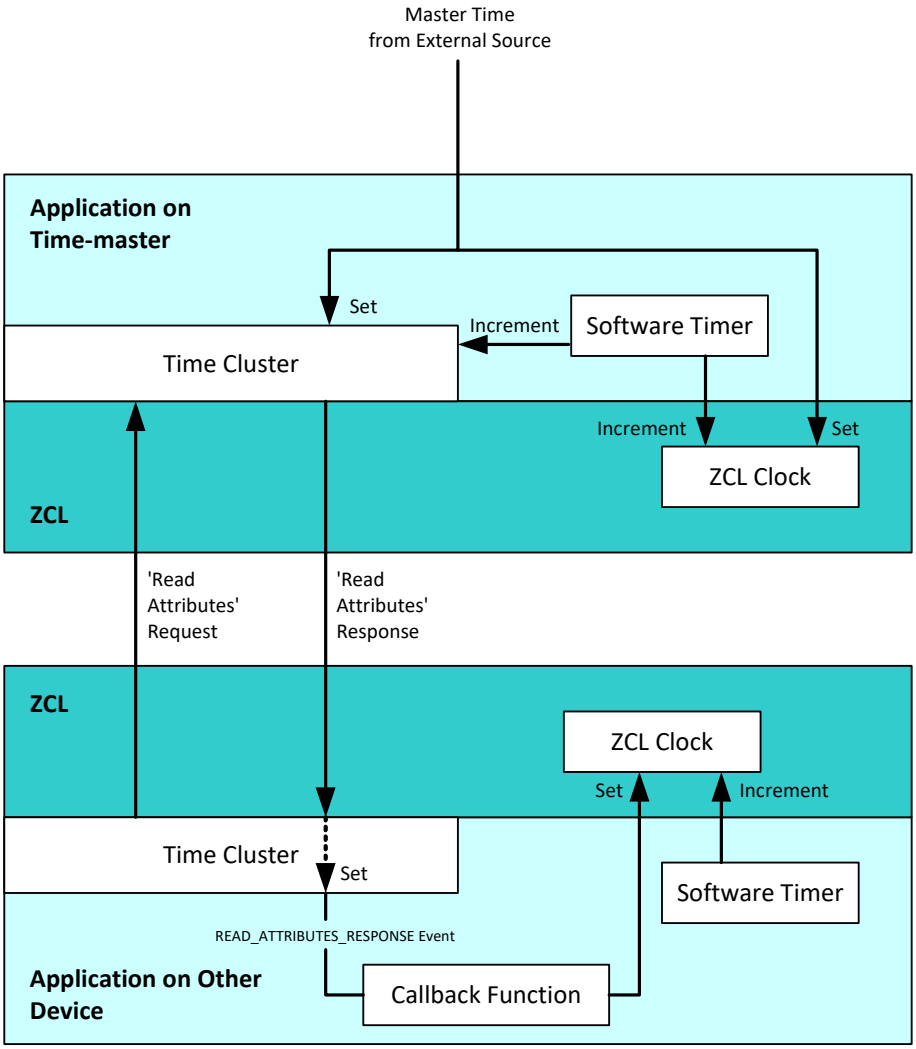
On all other devices, these times are initialized by the local application by synchronizing with the time-master (see Section 18.5.2). The ZCL time is subsequently maintained using a local one-second timer and both times are occasionally re-synchronized with the time-master (see Section 18.5.3).

synchronization with the time-master is normally performed via the Time cluster.

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

The diagram in Figure 4 below provides an overview of the time initialization and synchronization processes described in the sub-sections that follow.

Table 38. Time Initialization and Synchronization



18.5.1 Initialising and Maintaining Master Time

The time-master must initially obtain a master time from an external source. The application on the time-master 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 device within the ZigBee network.*

If the time-master has also obtained time-zone and daylight saving information (or has been pre-programmed with this information), its 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 can then be used to provide time-zone and daylight saving information to other devices (see Section 18.3).

Note: *The time-master can prevent other devices from attempting to read its Time cluster attributes before the time has been set - the initialization of the master time should be done after registering the endpoint for the device and before starting the ZigBee PRO stack.*

The ZCL time and the `utctTime` attribute are subsequently incremented from a local one-second timer, as follows. On expiration of the timer, an event is generated (from the hardware/software timer that drives the one-second timer), which causes a ZCL user task to be activated. The event is initially handled by this task as described in Section 3.2, 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.

Both the ZCL time and the `utctTime` attribute must also be updated by the application when an update of the master time is received.

18.5.2 Initial Synchronisation of Devices

It is the responsibility of the application on a ZigBee PRO device to perform time-synchronisation with the time-master. The application can remotely read the Time cluster attributes from the time-master by calling the function **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 a ZCL user task to be activated. The event is initially handled by this task as described in Section 3.2, 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 Chapter 3), 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()**

Note: *When a device attempts to time-synchronise with the time-master, 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 time-master. 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 18.5.3.

18.5.3 Re-synchronisation of Devices

After the initialization described in Section 18.5.2, the ZCL time must be updated by the application on each one-second tick of the local timer. The ZCL time is updated from the timer in the same way as described in Section 18.4.

Due to the inaccuracy of the local one-second timer, the ZCL time is likely to lose synchronisation with the time on the time-master. It will therefore be necessary to occasionally re-synchronise the local ZCL

time with the time-master - the `utctTime` attribute value is also updated at the same time. A device can re-synchronise with the time-master by first remotely reading the `utctTime` attribute using the function **eZCL_SendReadAttributesRequest()**. On receiving the 'read attributes' response from the time-master, the operations performed are the same as those described for initial synchronisation in Section 18.5.2.

18.6 Time Event

The Time cluster does not have any events of its own, but the ZCL includes one time-related event: E_ZCL_CBET_TIMER. For this event, the `eEventType` field of the `tsZCL_CallBackEvent` structure (see Section 3.1) is set to E_ZCL_CBET_TIMER.

The application may need to generate this event, as indicated in Section 3.2.

18.7 Functions

The following time-related functions are provided in the NXP implementation of the ZCL:

1. eCLD_TimeCreateTime
2. vZCL_SetUTCTime
3. u32ZCL_GetUTCTime
4. bZCL_GetTimeHasBeenSynchronised
5. vZCL_ClearTimeHasBeenSynchronised

Note: *The time used in the Time cluster and in the ZCL is a UTC (Co-ordinated Universal Time) type* ***UTCTime****, which is defined in the ZigBee Specification as follows: "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"*

18.7.1 eCLD_TimeCreateTime

```
teZCL_Status eCLD_TimeCreateTime(
    tsZCL_ClusterInstance *psClusterInstance,
    bool_t bIsServer,
    tsZCL_ClusterDefinition *psClusterDefinition,
    void *pvEndPointSharedStructPtr,
    uint8 *pu8AttributeControlBits);
```

Description

This function creates an instance of the Time cluster on the local endpoint. The cluster instance 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 Time cluster instance on the endpoint, but instances of other clusters may also be created on the same endpoint by calling their corresponding creation functions.

Note: *This function must not be called for an endpoint on which a standard ZigBee device (e.g. Simple Sensor) will be used. In this case, the device and its supported clusters must be registered on the endpoint using the relevant device registration function.*

When used, this function must be the first Time cluster function called in the application, and must be called after the stack has been started and after the ZCL has been initialized.

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 Time cluster.

The function initializes the array elements to zero.

Parameters

- *psClusterInstance*: Pointer to structure containing information about the cluster instance to be created (see Section 6.1.16). This structure is updated by the function by initializing individual structure fields.
- *bIsServer*: Type of cluster instance (server or client) to be created:
  - TRUE - server
  - FALSE - client
- *psClusterDefinition*: Pointer to structure indicating the type of cluster to be created (see Section 6.1.2). In this case, this structure must contain the details of the Time cluster. This parameter can refer to a pre-filled structure called `sCLD_Time` which is provided in the **Time.h** file.
- *pvEndPointSharedStructPtr*: Pointer to the shared structure used for attribute storage. This parameter should be the address of the structure of type `tsCLD_Time` which defines the attributes of Time cluster. The function initializes the attributes with default values.
- *pu8AttributeControlBits*: Pointer to an array of **uint8** values, with one element for each attribute in the cluster (see above).

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_INVALID_VALUE

18.7.2 vZCL_SetUTCTime

```
void vZCL_SetUTCTime(uint32 u32UTCTime);
```

Description

This function sets the current time (UTC) that is stored in the ZCL ('ZCL time').

The application may call this function, for example, when a time update has been received (e.g. via the Time or Price cluster).

Note that this function does not update the time in the Timer cluster - if required, the application must do this by writing to the `tsCLD_Time` structure (see Section 18.2).

Parameters

u32UTCTime The current time (UTC) to be set, in seconds

Returns

None

18.7.3 u32ZCL_GetUTCTime

```
uint32 u32ZCL_GetUTCTime(void);
```

Description

This function obtains the current time (UTC) that is stored in the ZCL (‘ZCL time’).

Parameters

None

Returns

The current time (UTC), in seconds, obtained from the ZCL

18.7.4 bZCL_GetTimeHasBeenSynchronised

```
bool_t bZCL_GetTimeHasBeenSynchronised(void);
```

Description

This function queries whether the ZCL time on the device has been synchronized.

The clock is considered to be unsynchronized at start-up and is synchronized following a call to **vZCL_SetUTCtime()**. The ZCL time must be synchronized before using the time-related functions of other clusters.

Parameters

None

Returns

TRUE if the local ZCL time has been synchronized, otherwise FALSE

18.7.5 vZCL_ClearTimeHasBeenSynchronised

```
void vZCL_ClearTimeHasBeenSynchronised(void);
```

Description

This function is used to notify the ZCL that the local ZCL time may no longer be accurate.

Parameters

None

Returns

None

18.8 Return codes

The time-related functions use the ZCL return codes defined in Section 7.2.

18.9 Enumerations

18.9.1 teCLD_TM_AttributeID

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

```
typedef enum
{
 E_CLD_TIME_ATTR_ID_TIME  = 0x0000, /* Mandatory */
 E_CLD_TIME_ATTR_ID_TIME_STATUS, /* Mandatory */
 E_CLD_TIME_ATTR_ID_TIME_ZONE,
 E_CLD_TIME_ATTR_ID_DST_START,
 E_CLD_TIME_ATTR_ID_DST_END,
 E_CLD_TIME_ATTR_ID_DST_SHIFT,
 E_CLD_TIME_ATTR_ID_STANDARD_TIME,
 E_CLD_TIME_ATTR_ID_LOCAL_TIME,
 E_CLD_TIME_ATTR_ID_LAST_SET_TIME,
 E_CLD_TIME_ATTR_ID_VALID_UNTIL_TIME
} teCLD_TM_AttributeID;
```

18.10 Compile-time Options

To enable the Time cluster in the code to be built, it is necessary to add the following to the **zcl_options.h** file:

```
#define CLD_TIME
```

In addition, to include the software for a cluster client or server or both, it is necessary to add one or both of the following to the same file:

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

The Time cluster contains macros that may be optionally specified at compile-time by adding some or all of the following lines to the **zcl_options.h** file.

Add this line to enable the optional Time Zone attribute

```
#define CLD_TIME_ATTR_TIME_ZONE
```

Add this line to enable the optional DST Start attribute

```
#define CLD_TIME_ATTR_DST_START
```

Add this line to enable the optional DST End attribute

```
#define CLD_TIME_ATTR_DST_END
```

Add this line to enable the optional DST Shift attribute

```
#define CLD_TIME_ATTR_DST_SHIFT
```

Add this line to enable the optional Standard Time attribute

```
#define CLD_TIME_ATTR_STANDARD_TIME
```

Add this line to enable the optional Local Time attribute

```
#define CLD_TIME_ATTR_LOCAL_TIME
```

Note: *Some attributes must always be enabled together - for example, if daylight saving is to be implemented then CLD_TIME_ATTR_DST_START, CLD_TIME_ATTR_DST_END and CLD_TIME_ATTR_DST_SHIFT must all be included in the* ***zcl_options.h*** *file.*

Add this line to define the value (n) of the Cluster Revision attribute:

```
#define CLD_TIME_CLUSTER_REVISION <n>
```

The default value is 1, which corresponds to the revision of the cluster in the ZCL r6 specification (see Section 2.4).

19 Input and Output Clusters

This chapter details the following input and output clusters:

- Analogue Input (Basic) - see Section 19.1
- Analogue Output (Basic) - see Section 19.2
- Binary Input (Basic) - see Section 19.3
- Binary Output (Basic) - see Section 19.4
- Multistate Input (Basic) - see Section 19.5
- Multistate Output (Basic) - see Section 19.6

19.1 Analogue Input (Basic)

This chapter describes the Analogue Input (Basic) cluster, which provides an interface for accessing an analogue measurement.

The Analogue Input (Basic) cluster has a Cluster ID of 0x000C.

19.1.1 Overview

The Analogue Input (Basic) cluster provides an interface for accessing an analogue measurement and its associated characteristics. It is typically used in a sensor that measures an analogue physical quantity.

To use the functionality of this cluster, you must include the file **AnalogInputBasic.h** in your application and enable the cluster by defining CLD_ANALOG_INPUT_BASIC in the **zcl_options.h** file.

An Analogue Input (Basic) cluster instance can act as either a client or a server. The inclusion of the client or server software must be pre-defined in the application's compile-time options (in addition, if the cluster is to reside on a custom endpoint then the role of client or server must also be specified when creating the cluster instance).

The compile-time options for the Analogue Input (Basic) cluster are fully detailed in Section 19.1.6.

19.1.2 Analogue Input (Basic) Structure and Attributes

The structure definition for the Analogue Input (Basic) cluster is:

```
typedef struct
{
#ifdef ANALOG_INPUT_BASIC_SERVER
#ifdef CLD_ANALOG_INPUT_BASIC_ATTR_DESCRIPTION
    tsZCL_CharacterString       sDescription;
    zuint8    au8Description[16];
#endif
#ifdef CLD_ANALOG_INPUT_BASIC_ATTR_MAX_PRESENT_VALUE
    zsingle   fMaxPresentValue;
#endif
#ifdef CLD_ANALOG_INPUT_BASIC_ATTR_MIN_PRESENT_VALUE
    zsingle   fMinPresentValue;
#endif
    zbool     bOutOfService;
    zsingle   fPresentValue;
#ifdef CLD_ANALOG_INPUT_BASIC_ATTR_RELIABILITY
    zenum8    u8Reliability;
#endif
#ifdef CLD_ANALOG_INPUT_BASIC_ATTR_RESOLUTION
    zsingle   fResolution;
#endif
    zbmap8    u8StatusFlags;
```

```
#ifdef CLD_ANALOG_INPUT_BASIC_ATTR_ENGINEERING_UNITS
    zenum16   u16EngineeringUnits;
#endif
#ifdef CLD_ANALOG_INPUT_BASIC_ATTR_APPLICATION_TYPE
    zuint32   u32ApplicationType;
#endif
#ifdef CLD_ANALOG_INPUT_BASIC_ATTR_ATTRIBUTE_REPORTING_STATUS
    zenum8    u8AttributeReportingStatus;
#endif
#endif
    zuint16   u16ClusterRevision;
} tsCLD_AnalogInputBasic;
```

- The following optional pair of attributes are used to store a human readable description of the usage of the analogue input (for example, "Kitchen Temp"):
  - `sDescription` is a `tsZCL_CharacterString` structure (see Section 6.1.14) for a string of up to 16 characters representing the description
  - `au8Description[16]` is a byte-array which contains the character data bytes representing the description
  - `fMaxPresentValue` is an optional attribute which indicates the highest analogue input value that can be reliably obtained and stored in the `fPresentValue` attribute.
- `fMinPresentValue` is an optional attribute which indicates the lowest analogue input value that can be reliably obtained and stored in the `fPresentValue` attribute.
- `bOutOfService` is a mandatory attribute which indicates whether the analogue input is currently in or out of service:
  - TRUE: Out of service
  - FALSE In service

    If this attribute is set to TRUE, the `fPresentValue` attribute will not be updated to contain the current value of the input.
- `fPresentValue` is a mandatory attribute representing the latest analogue input value (this attribute is updated when the analogue input is re-sampled).
- `u8Reliability` is an optional attribute which indicates whether the value reported through `fPresentValue` is reliable or why it might be unreliable:
  - E_CLD_ANALOG_INPUT_BASIC_RELIABILITY_NO_FAULT_DETECTED
  - E_CLD_ANALOG_INPUT_BASIC_RELIABILITY_NO_SENSOR
  - E_CLD_ANALOG_INPUT_BASIC_RELIABILITY_OVER_RANGE
  - E_CLD_ANALOG_INPUT_BASIC_RELIABILITY_UNDER_RANGE
  - E_CLD_ANALOG_INPUT_BASIC_RELIABILITY_OPEN_LOOP
  - E_CLD_ANALOG_INPUT_BASIC_RELIABILITY_SHORTED_LOOP
  - E_CLD_ANALOG_INPUT_BASIC_RELIABILITY_NO_OUTPUT
  - E_CLD_ANALOG_INPUT_BASIC_RELIABILITY_UNRELIABLE_OTHER
  - E_CLD_ANALOG_INPUT_BASIC_RELIABILITY_PROCESS_ERROR
  - E_CLD_ANALOG_INPUT_BASIC_RELIABILITY_CONFIGURATION_ERROR
- `fResolution` is an optional attribute which indicates the smallest detectable change in the analogue input value that will result in an update of the attribute fPresentValue.
- `u8StatusFlags` is a mandatory attribute which is a bitmap representing the following status flags:

Bits	Name	Description
0	In Alarm	Reserved - unused for Analogue Input (Basic) cluster
1	Fault	• 1: Optional attribute `u8Reliability` is used and does not have a value of NO_FAULT_DETECTED

Bits	Name	Description
		• 0: Otherwise
2	Overridden	• 1: Cluster has been overridden by a local mechanism (`fPresentValue` and `u8Reliability` will not track input) • 0: Otherwise
3	Out Of Service	• 1: Attribute `bOutOfService` is set to TRUE • 0: Otherwise
4-7	-	Reserved

- `u16EngineeringUnits` is an optional attribute which indicates the physical unit of measure for the analogue input value recorded in the attribute `fPresentValue`. The values 0x0000 to 0x00FE are used to represent the units specified in Clause 21 of the BACnet standard. The value 0x00FF represents 'other' unit, and the values 0x0100 to 0xFFFF are for proprietary use. If the attribute `u32ApplicationType` is used and specifies an application type with an associated unit of measure, this unit will take precedence over the one specified in `u16EngineeringUnits`.
- `u32ApplicationType` is an optional attribute which is a bitmap representing the application type, as follows:

Bits	Field Name	Description
0-15	Index	Specific application usage (for example,Boiler Entering Temperature). There is a set of possible usages for each value of Type (see below). For these lists, refer to the attribute description in the ZCL Specification.
16-23	Type	Physical quantity measured (for example,Temperature). For the Analogue Input cluster, this can be a value in the range 0x00 to 0x0E. For the corresponding quantities, refer to the attribute description in the ZCL Specification.
24-31	Group	Identifier for the cluster that this attribute is part of (not the Cluster ID). For the Analogue Input cluster, this is 0x00.

- `u8AttributeReportingStatus` is an optional attribute that should be enabled when attribute reporting is used for the cluster (see Section 2.3.5). The value of this attribute indicates whether there are attribute reports still pending (0x00) or the attribute reports are complete (0x01) - all other values are reserved. This attribute is also described in Section 2.4.
- `u16ClusterRevision` is a mandatory attribute that specifies the revision of the cluster specification on which this cluster instance is based. The cluster specification in the ZCL r6 corresponds to a cluster revision of 1. The value is incremented by one for each subsequent revision of the cluster specification. This attribute is also described in Section 2.4.

19.1.3 Attributes for Default Reporting

The following attributes of the Analogue Input (Basic) cluster can be selected for default reporting:

```
fPresentValue
```

- `u8AttributeReportingStatus`

Attribute reporting (including default reporting) is described in Appendix B. Enabling reports for these attributes is described in Appendix B.3.6.

19.1.4 Functions

The following Analogue Input (Basic) cluster function is provided in the NXP implementation of the ZCL:

eCLD_AnalogInputBasicCreateAnalogInputBasic

The cluster attributes can be accessed using the general attribute read/write functions, as described in Section 2.3.

19.1.4.1 eCLD_AnalogInputBasicCreateAnalogInputBasic

```
teZCL_Status eCLD_AnalogInputBasicCreateAnalogInputBasic(
  tsZCL_ClusterInstance *psClusterInstance,
  bool_t bIsServer,
  tsZCL_ClusterDefinition *psClusterDefinition,
  void *pvEndPointSharedStructPtr,
  uint8 *pu8AttributeControlBits);
```

Description

This function creates an instance of the Analogue Input (Basic) 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 an Analogue Input (Basic) cluster instance on the endpoint, but instances of other clusters may also be created on the same endpoint by calling their corresponding creation functions.

Note: *This function must not be called for an endpoint on which a standard ZigBee device is used. In this case, the device and its supported clusters must be registered on the endpoint using the relevant device registration function.*

When used, this function must be called after the stack has been started and after the ZCL has been initialized.

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 Analog Input Basic cluster.

The function initializesthe array elements to zero.

Parameters

- *psClusterInstance*: Pointer to structure containing information about the cluster instance to be created (see Section 6.1.16). This structure is updated by the function by initializing individual structure fields.
- *bIsServer*: Type of cluster instance (server or client) to be created:
- TRUE - server
- FALSE - client
- *psClusterDefinition*: Pointer to structure indicating the type of cluster to be created (see Section 6.1.2). In this case, this structure must contain the details of the Analogue Input (Basic) cluster. This parameter can refer to a pre-filled structure called `sCLD_AnalogInputBasic` which is provided in the **AnalogInputBasic.h** file.
- *pvEndPointSharedStructPtr*: Pointer to the shared structure used for attribute storage. This parameter should be the address of the structure of type `tsCLD_AnalogInputBasic` which defines the attributes of Analogue Input (Basic) cluster. The function initializes the attributes with default values.
- *pu8AttributeControlBits*: Pointer to an array of **uint8** values, with one element for each attribute in the cluster (see above).

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_INVALID_VALUE

19.1.5 Enumerations

19.1.5.1 teCLD_AnalogInputBasicCluster_AttrID

The following structure contains the enumerations used to identify the attributes of the Analogue Input (Basic) cluster.

```
typedef enum
{
    E_CLD_ANALOG_INPUT_BASIC_ATTR_ID_DESCRIPTION,
    E_CLD_ANALOG_INPUT_BASIC_ATTR_ID_MAX_PRESENT_VALUE,
    E_CLD_ANALOG_INPUT_BASIC_ATTR_ID_MIN_PRESENT_VALUE,
    E_CLD_ANALOG_INPUT_BASIC_ATTR_ID_OUT_OF_SERVICE,
    E_CLD_ANALOG_INPUT_BASIC_ATTR_ID_PRESENT_VALUE,
    E_CLD_ANALOG_INPUT_BASIC_ATTR_ID_RELIABILITY,
    E_CLD_ANALOG_INPUT_BASIC_ATTR_ID_RESOLUTION,
    E_CLD_ANALOG_INPUT_BASIC_ATTR_ID_STATUS_FLAGS,
    E_CLD_ANALOG_INPUT_BASIC_ATTR_ID_ENGINEERING_UNITS,
    E_CLD_ANALOG_INPUT_BASIC_ATTR_ID_APPLICATION_TYPE,
} teCLD_AnalogInputBasicCluster_AttrID;
```

19.1.5.2 teCLD_AnalogInputBasic_Reliability

The following structure contains the enumerations used to report the value of the `u8Reliability` attribute (see Section 19.1.2).

```
typedef enum
{
    E_CLD_ANALOG_INPUT_BASIC_RELIABILITY_NO_FAULT_DETECTED,
    E_CLD_ANALOG_INPUT_BASIC_RELIABILITY_NO_SENSOR,
    E_CLD_ANALOG_INPUT_BASIC_RELIABILITY_OVER_RANGE,
    E_CLD_ANALOG_INPUT_BASIC_RELIABILITY_UNDER_RANGE,
    E_CLD_ANALOG_INPUT_BASIC_RELIABILITY_OPEN_LOOP,
    E_CLD_ANALOG_INPUT_BASIC_RELIABILITY_SHORTED_LOOP,
    E_CLD_ANALOG_INPUT_BASIC_RELIABILITY_NO_OUTPUT,
    E_CLD_ANALOG_INPUT_BASIC_RELIABILITY_UNRELIABLE_OTHER,
    E_CLD_ANALOG_INPUT_BASIC_RELIABILITY_PROCESS_ERROR,
    E_CLD_ANALOG_INPUT_BASIC_RELIABILITY_CONFIGURATION_ERROR
}teCLD_AnalogInputBasic_Reliability;
```

19.1.6 Compile-time Options

To enable the Analogue Input (Basic) cluster in the code to be built, it is necessary to add the following to the **zcl_options.h** file:

```
#define CLD_ANALOG_INPUT_BASIC
```

In addition, to include the software for a cluster client or server or both, it is necessary to add one or both of the following to the same file:

```
#define ANALOG_INPUT_BASIC_CLIENT
#define ANALOG_INPUT_BASIC_SERVER
```

Optional Attributes

The optional attributes for the Analogue Input (Basic) cluster (see Section 19.1.2) are enabled by defining:

- CLD_ANALOG_INPUT_BASIC_ATTR_DESCRIPTION
- CLD_ANALOG_INPUT_BASIC_ATTR_MAX_PRESENT_VALUE
- CLD_ANALOG_INPUT_BASIC_ATTR_MIN_PRESENT_VALUE
- CLD_ANALOG_INPUT_BASIC_ATTR_RELIABILITY
- CLD_ANALOG_INPUT_BASIC_ATTR_RESOLUTION
- CLD_ANALOG_INPUT_BASIC_ATTR_ENGINEERING_UNITS
- CLD_ANALOG_INPUT_BASIC_ATTR_APPLICATION_TYPE
- CLD_ANALOG_INPUT_BASIC_ATTR_ATTRIBUTE_REPORTING_STATUS

Global Attributes

Add this line to enable the optional Attribute Reporting Status attribute:

```
#define CLD_ANALOG_INPUT_BASIC_ATTR_ID_ATTRIBUTE_REPORTING_STATUS
```

Add this line to define the value (n) of the Cluster Revision attribute:

```
#define CLD_ANALOG_INPUT_BASIC_CLUSTER_REVISION <n>
```

The default value is 1, which corresponds to the revision of the cluster in the ZCL r6 specification (see Section 2.4).

19.2 Analogue Output (Basic)

This chapter describes the Analogue Output (Basic) cluster, which provides an interface for setting the value of an analogue output.

The Analogue Output (Basic) cluster has a Cluster ID of 0x000D.

19.2.1 Overview

The Analogue Input (Basic) cluster provides an interface for setting the value of an analogue output and accessing its associated characteristics. It is typically used in a a controller that outputs an analogue control signal.

To use the functionality of this cluster, you must include the file **AnalogOutputBasic.h** in your application and enable the cluster by defining CLD_ANALOG_OUTPUT_BASIC in the **zcl_options.h** file.

An Analogue Output (Basic) cluster instance can act as either a client or a server. The inclusion of the client or server software must be pre-defined in the application's compile-time options (in addition, if the cluster is to reside on a custom endpoint then the role of client or server must also be specified when creating the cluster instance).

The compile-time options for the Analogue Output (Basic) cluster are fully detailed in Section 19.2.6.

19.2.2 Analogue Output (Basic) Structure and Attributes

The structure definition for the Analogue Output (Basic) cluster is:

```
 typedef struct
 {
 #ifdef ANALOG_OUTPUT_BASIC_SERVER
 #ifdef CLD_ANALOG_OUTPUT_BASIC_ATTR_DESCRIPTION
  tsZCL_CharacterString  sDescription;
  uint8  au8Description[16];
#endif
#ifdef  CLD_ANALOG_OUTPUT_BASIC_ATTR_MAX_PRESENT_VALUE
  zsingle       fMaxPresentValue;
#endif
#ifdef CLD_ANALOG_OUTPUT_BASIC_ATTR_MIN_PRESENT_VALUE
  zsingle       fMinPresentValue;
#endif
  zbool         bOutOfService;
  zsingle       fPresentValue;
#ifdef CLD_ANALOG_OUTPUT_BASIC_ATTR_RELIABILITY
  zenum8        u8Reliability;
#endif
#ifdef CLD_ANALOG_OUTPUT_BASIC_ATTR_RELINQUISH_DEFAULT
  zsingle       fRelinquishDefault;
#endif
#ifdef CLD_ANALOG_OUTPUT_BASIC_ATTR_RESOLUTION
  zsingle       fResolution;
#endif
  zbmap8        u8StatusFlags;
#ifdef CLD_ANALOG_OUTPUT_BASIC_ATTR_ENGINEERING_UNITS
  zenum16       u16EngineeringUnits;
#endif
#ifdef CLD_ANALOG_OUTPUT_BASIC_ATTR_APPLICATION_TYPE
  zuint32       u32ApplicationType;
#endif
#ifdef CLD_ANALOG_OUTPUT_BASIC_ATTR_ATTRIBUTE_REPORTING_STATUS
  zenum8        u8AttributeReportingStatus;
#endif
#endif
  zuint16       u16ClusterRevision;
} tsCLD_AnalogOutputBasic;
```

- The following optional pair of attributes store a human readable description of the usage of the analogue output (for example, "Fan Speed"):
  - `sDescription` is a `tsZCL_CharacterString` structure (see Section 6.1.14) for a string of up to 16 characters representing the description:
  - `au8Description[16]` is a byte-array which contains the character data bytes representing the description
  - fMaxPresentValue is an optional attribute which indicates the highest analogue output value that can be reliably used and stored in the fPresentValue attribute.
  - `fMinPresentValue` is an optional attribute which indicates the lowest analogue output value that can be reliably used and stored in the `fPresentValue` attribute.
- `bOutOfService` is a mandatory attribute which indicates whether the analogue output is currently in or out of service:
  - TRUE: Out of service
  - FALSE: In service

If this attribute is set to TRUE, the `fPresentValue` attribute will not be used to control the physical analogue output.

- `fPresentValue` is a mandatory attribute representing the latest analogue output value (this attribute is used to control the physical analogue output).
- `u8Reliability` is an optional attribute which indicates whether the value contained in `fPresentValue` is reliable or why it might be unreliable:
  - E_CLD_ANALOG_INPUT_BASIC_RELIABILITY_NO_FAULT_DETECTED
  - E_CLD_ANALOG_INPUT_BASIC_RELIABILITY_OVER_RANGE
  - E_CLD_ANALOG_INPUT_BASIC_RELIABILITY_UNDER_RANGE
  - E_CLD_ANALOG_INPUT_BASIC_RELIABILITY_OPEN_LOOP
  - E_CLD_ANALOG_INPUT_BASIC_RELIABILITY_SHORTED_LOOP
  - E_CLD_ANALOG_INPUT_BASIC_RELIABILITY_UNRELIABLE_OTHER
  - E_CLD_ANALOG_INPUT_BASIC_RELIABILITY_PROCESS_ERROR
  - E_CLD_ANALOG_INPUT_BASIC_RELIABILITY_CONFIGURATION_ERROR
- `fRelinquishDefault` is an optional attribute representing the default value to be used for fPresentValue when the supplied value is invalid.
- `fResolution` is an optional attribute which indicates the smallest change in the analogue output value that will cause the application to update the attribute `fPresentValue`.
- `u8StatusFlags` is a mandatory attribute which is a bitmap representing the following status flags:

Bits	Name	Description
0	In Alarm	Reserved - unused for Analogue Output (Basic) cluster
1	Fault	• 1: Optional attribute `u8Reliability` is used and does not have a value of NO_FAULT_DETECTED • 0: Otherwise
2	Overridden	• 1: Cluster has been over-ridden by a local mechanism (`fPresentValue` and `u8Reliability` will not track input) • 0: Otherwise
3	Out Of Service	• 1: Attribute `bOutOfService` is set to TRUE • 0: Otherwise
4-7	-	Reserved

- `u16EngineeringUnits` is an optional attribute which indicates the physical unit of measure for the analogue output value recorded in the attribute `fPresentValue`. The values 0x0000 to 0x00FE are used to represent the units specified in Clause 21 of the BACnet standard. The value 0x00FF represents 'other' unit, and the values 0x0100 to 0xFFFF are for proprietary use. If the attribute `u32ApplicationType` is used and specifies an application type with an associated unit of measure, this unit will take precedence over the one specified in `u16EngineeringUnits`.
- `u32ApplicationType` is an optional attribute which is a bitmap representing the application type, as follows:

Bits	Field Name	Description
0-15	Index	Specific application usage (for example, Fan Speed). There is a set of possible usages for each value of Type (see below). For these lists, refer to the attribute description in the ZCL Specification.
16-23	Type	Physical quantity controlled (for example, Rotational Speed). For the Analogue Output cluster, this can be a value in the range 0x00 to 0x0E. For the corresponding quantities, refer to the attribute description in the ZCL Specification.

Bits	Field Name	Description
24-31	Group	Identifier for the cluster that this attribute is part of (not the Cluster ID). For the Analogue Output cluster, this is 0x01.

- `u8AttributeReportingStatus` is an optional attribute that should be enabled when attribute reporting is used for the cluster (see Section 2.3.5). The value of this attribute indicates whether there are attribute reports still pending (0x00) or the attribute reports are complete (0x01) - all other values are reserved. This attribute is also described in Section 2.4.
- `u16ClusterRevision` is a mandatory attribute that specifies the revision of the cluster specification on which this cluster instance is based. The cluster specification in the ZCL r6 corresponds to a cluster revision of `1`. The value is incremented by one for each subsequent revision of the cluster specification. This attribute is also described in Section 2.4.

19.2.3 Attributes for Default Reporting

The following attributes of the Analogue Output (Basic) cluster can be selected for default reporting:

```
fPresentValue
u8AttributeReportingStatus
```

Attribute reporting (including default reporting) is described in Appendix B. Enabling reports for these attributes is described in Appendix B.3.6.

19.2.4 Functions

The following Analogue Output (Basic) cluster function is provided in the NXP implementation of the ZCL:

- eCLD_AnalogOutputBasicCreateAnalogOutputBasic

The cluster attributes can be accessed using the general attribute read/write functions, as described in Section 2.3.

19.2.4.1 eCLD_AnalogOutputBasicCreateAnalogOutputBasic

```
teZCL_Status eCLD_AnalogOutputBasicCreateAnalogOutputBasic(
        tsZCL_ClusterInstance *psClusterInstance,
        bool_t bIsServer,
        tsZCL_ClusterDefinition *psClusterDefinition,
        void *pvEndPointSharedStructPtr,
        uint8 *pu8AttributeControlBits);
```

Description

This function creates an instance of the Analogue Output (Basic) 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 an Analogue Output (Basic) cluster instance on the endpoint, but instances of other clusters may also be created on the same endpoint by calling their corresponding creation functions.

Note: *This function must not be called for an endpoint on which a standard ZigBee device is used. In this case, the device and its supported clusters must be registered on the endpoint using the relevant device registration function.*

When used, this function must be called after the stack has been started and after the ZCL has been initialized.

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 Analog Output Basic cluster.

The function initializes the array elements to zero.

Parameters

psClusterInstance Pointer to structure containing information about the cluster instance to be created (see Section 6.1.16). This structure is updated by the function by initialising individual structure fields.

bIsServer Type of cluster instance (server or client) to be created:

TRUE - server

FALSE - client

psClusterDefinition Pointer to structure indicating the type of cluster to be created (see Section 6.1.2). In this case, this structure must contain the details of the Analogue Output (Basic) cluster. This parameter can refer to a pre-filled structure called `sCLD_AnalogOutputBasic` which is provided in the **AnalogOutputBasic.h** file.

pvEndPointSharedStructPtr Pointer to the shared structure used for attribute storage. This parameter should be the address of the structure of type `tsCLD_AnalogOutputBasic` which defines the attributes of Analogue Output (Basic) cluster. The function initializes the attributes with default values.

pu8AttributeControlBits Pointer to an array of **uint8** values, with one element for each attribute in the cluster (see above).

Returns

E_ZCL_SUCCESS
E_ZCL_FAIL
E_ZCL_ERR_PARAMETER_NULL
E_ZCL_ERR_INVALID_VALUE

19.2.5 Enumerations

19.2.5.1 teCLD_AnalogOutputBasicCluster_AttrID

The following structure contains the enumerations used to identify the attributes of the Analogue Output (Basic) cluster.

```
typedef enum
{
    E_CLD_ANALOG_OUTPUT_BASIC_ATTR_ID_DESCRIPTION,
    E_CLD_ANALOG_OUTPUT_BASIC_ATTR_ID_MAX_PRESENT_VALUE,
    E_CLD_ANALOG_OUTPUT_BASIC_ATTR_ID_MIN_PRESENT_VALUE,
    E_CLD_ANALOG_OUTPUT_BASIC_ATTR_ID_OUT_OF_SERVICE,
    E_CLD_ANALOG_OUTPUT_BASIC_ATTR_ID_PRESENT_VALUE,
    E_CLD_ANALOG_OUTPUT_BASIC_ATTR_ID_RELIABILITY,
    E_CLD_ANALOG_OUTPUT_BASIC_ATTR_ID_RELINQUISH_DEFAULT,
    E_CLD_ANALOG_OUTPUT_BASIC_ATTR_ID_RESOLUTION,
    E_CLD_ANALOG_OUTPUT_BASIC_ATTR_ID_STATUS_FLAGS,
    E_CLD_ANALOG_OUTPUT_BASIC_ATTR_ID_ENGINEERING_UNITS,
```

```
    E_CLD_ANALOG_OUTPUT_BASIC_ATTR_ID_APPLICATION_TYPE,
} teCLD_AnalogOutputBasicCluster_AttrID;
```

19.2.5.2 teCLD_AnalogOutputBasic_Reliability

The following structure contains the enumerations used to report the value of the `u8Reliability` attribute (see Section 19.2.2).

```
typedef enum
{
   E_CLD_ANALOG_OUTPUT_BASIC_RELIABILITY_NO_FAULT_DETECTED,
   E_CLD_ANALOG_OUTPUT_BASIC_RELIABILITY_OVER_RANGE,
   E_CLD_ANALOG_OUTPUT_BASIC_RELIABILITY_UNDER_RANGE,
   E_CLD_ANALOG_OUTPUT_BASIC_RELIABILITY_OPEN_LOOP,
   E_CLD_ANALOG_OUTPUT_BASIC_RELIABILITY_SHORTED_LOOP,
   E_CLD_ANALOG_OUTPUT_BASIC_RELIABILITY_UNRELIABLE_OTHER,
   E_CLD_ANALOG_OUTPUT_BASIC_RELIABILITY_PROCESS_ERROR,
   E_CLD_ANALOG_OUTPUT_BASIC_RELIABILITY_CONFIGURATION_ERROR
}teCLD_AnalogOutputBasic_Reliability;
```

19.2.6 Compile-time options

To enable the Analogue Output (Basic) cluster in the code to be built, it is necessary to add the following to the **zcl_options.h** file:

```
#define CLD_ANALOG_OUTPUT_BASIC
```

In addition, to include the software for a cluster client or server or both, it is necessary to add one or both of the following to the same file:

```
#define ANALOG_OUTPUT_BASIC_CLIENT
#define ANALOG_OUTPUT_BASIC_SERVER
```

Optional Attributes

The optional attributes for the Analogue Output (Basic) cluster (see Section 19.2.2) are enabled by defining:

- CLD_ANALOG_OUTPUT_BASIC_ATTR_DESCRIPTION
- CLD_ANALOG_OUTPUT_BASIC_ATTR_MAX_PRESENT_VALUE
- CLD_ANALOG_OUTPUT_BASIC_ATTR_MIN_PRESENT_VALUE
- CLD_ANALOG_OUTPUT_BASIC_ATTR_RELIABILITY
- CLD_ANALOG_OUTPUT_BASIC_ATTR_RELINQUISH_DEFAULT
- CLD_ANALOG_OUTPUT_BASIC_ATTR_RESOLUTION
- CLD_ANALOG_OUTPUT_BASIC_ATTR_ENGINEERING_UNITS
- CLD_ANALOG_OUTPUT_BASIC_ATTR_APPLICATION_TYPE
- CLD_ANALOG_OUTPUT_BASIC_ATTR_ATTRIBUTE_REPORTING_STATUS

Global Attributes

Add this line to enable the optional Attribute Reporting Status attribute:

```
#define CLD_ANALOG_OUTPUT_BASIC_ATTR_ID_ATTRIBUTE_REPORTING_STATUS
```

Add this line to define the value (n) of the Cluster Revision attribute:

```
#define CLD_ANALOG_OUTPUT_BASIC_CLUSTER_REVISION <n>
```

The default value is 1, which corresponds to the revision of the cluster in the ZCL r6 specification (see Section 2.4).

19.3 Binary Input (Basic) Cluster

This chapter describes the Binary Input (Basic) cluster, which provides an interface for accessing a binary (two-state) measurement.

The Binary Input (Basic) cluster has a Cluster ID of 0x000F.

19.3.1 Overview

The Binary Input (Basic) cluster provides an interface for accessing a binary measurement and its associated characteristics. It is typically used to implement a sensor that measures a two-state physical quantity.

To use the functionality of this cluster, you must include the file **BinaryInputBasic.h** in your application and enable the cluster by defining CLD_BINARY_INPUT_BASIC in the **zcl_options.h** file.

A Binary Input (Basic) cluster instance can act as either a client or a server. The inclusion of the client or server software must be pre-defined in the application's compile-time options (in addition, if the cluster is to reside on a custom endpoint then the role of client or server must also be specified when creating the cluster instance).

The compile-time options for the Binary Input (Basic) cluster are fully detailed in Section 19.3.6.

19.3.2 Binary Input (Basic) Structure and Attributes

The structure definition for the Binary Input (Basic) cluster is:

```
typedef struct
{
#ifdef BINARY_INPUT_BASIC_SERVER
#ifdef CLD_BINARY_INPUT_BASIC_ATTR_ACTIVE_TEXT
    tsZCL_CharacterString       sActiveText;
    uint8                       au8ActiveText[16];
#endif
#ifdef CLD_BINARY_INPUT_BASIC_ATTR_DESCRIPTION
    tsZCL_CharacterString       sDescription;
    uint8                       au8Description[16];
#endif
#ifdef CLD_BINARY_INPUT_BASIC_ATTR_INACTIVE_TEXT
    tsZCL_CharacterString       sInactiveText;
    uint8                       au8InactiveText[16];
#endif
    zbool                       bOutOfService;
#ifdef CLD_BINARY_INPUT_BASIC_ATTR_POLARITY
    zenum8                      u8Polarity;
#endif
    zbool                       bPresentValue;
#ifdef CLD_BINARY_INPUT_BASIC_ATTR_RELIABILITY
    zenum8                      u8Reliability;
#endif
    zbmap8                      u8StatusFlags;
#ifdef CLD_BINARY_INPUT_BASIC_ATTR_APPLICATION_TYPE
    zuint32                     u32ApplicationType;
```

```
#endif
#ifdef CLD_BINARY_INPUT_BASIC_ATTR_ATTRIBUTE_REPORTING_STATUS
    zenum8                      u8AttributeReportingStatus;
#endif
#endif
    zuint16                     u16ClusterRevision;
} tsCLD_BinaryInputBasic;
```

- The following optional pair of attributes are used to store a human readable description of the active state of a binary input (for example, "Window 3 open"):
  - `sActiveText` is a `tsZCL_CharacterString` structure (see Section 6.1.14) for a string of up to 16 characters representing the description
  - `au8ActiveText[16]` is a byte-array which contains the character data bytes representing the description

    If these attributes are used, the ‘Inactive Text’ attributes must also be used.
- The following optional pair of attributes are used to store a human readable description of the usage of the binary input (for example,"Window 3 status"):
  - `sDescription` is a `tsZCL_CharacterString` structure (see Section 6.1.14) for a string of up to 16 characters representing the description
  - `au8Description[16]` is a byte-array which contains the character data bytes representing the description
- The following optional pair of attributes are used to store a human readable description of the inactive state of a binary input (for example,"Window 3 closed"):
  - `sInactiveText` is a `tsZCL_CharacterString` structure (see Section 6.1.14) for a string of up to 16 characters representing the description
  - `au8InactiveText[16]` is a byte-array which contains the character data bytes representing the description

    If these attributes are used, the ‘Active Text’ attributes must also be used.
- `bOutOfService` is a mandatory attribute which indicates whether the binary input is currently in or out of service:
  - TRUE: Out of service
  - FALSE In service

    If this attribute is set to TRUE, the `bPresentValue` attribute will not be updated to contain the current value of the input.
- `u8Polarity` is an optional attribute which indicates the relationship between the value of the `bPresentValue` attribute and the physical state of the input:
  - E_CLD_ BINARY_INPUT_BASIC_POLARITY_NORMAL (0x00): The active (1) state of `bPresentValue` corresponds to the active/on state of the physical input
  - E_CLD_ BINARY_INPUT_BASIC_POLARITY_REVERSE (0x01): The active (1) state of `bPresentValue` corresponds to the inactive/off state of the physical input
- `bPresentValue` is a mandatory attribute representing the current state of the binary input (this attribute is updated when the input changes state):
  - TRUE: Input is in the ‘active’ state
  - FALSE: Input is in the ‘inactive’ state

    The interpretation of `bPresentValue` in relation to the physical state of the input is determined by the setting of the `u8Polarity` attribute.

    By default this attribute is read-only, but it becomes readable and writable when `bOutOfService` is set to TRUE.
- `u8Reliability` is an optional attribute which indicates whether the value reported through `bPresentValue` is reliable or why it might be unreliable:
  - E_CLD_ BINARY_INPUT_BASIC_RELIABILITY_NO_FAULT_DETECTED
  - E_CLD_ BINARY_INPUT_BASIC_RELIABILITY_NO_SENSOR

- E_CLD_ BINARY_INPUT_BASIC_RELIABILITY_OVER_RANGE
- E_CLD_ BINARY_INPUT_BASIC_RELIABILITY_UNDER_RANGE
- E_CLD_ BINARY_INPUT_BASIC_RELIABILITY_OPEN_LOOP
- E_CLD_ BINARY_INPUT_BASIC_RELIABILITY_SHORTED_LOOP
- E_CLD_ BINARY_INPUT_BASIC_RELIABILITY_NO_OUTPUT
- E_CLD_ BINARY_INPUT_BASIC_RELIABILITY_UNRELIABLE_OTHER
- E_CLD_ BINARY_INPUT_BASIC_RELIABILITY_PROCESS_ERROR
- E_CLD_ BINARY_INPUT_BASIC_RELIABILITY_CONFIGURATION_ERROR

- `u8StatusFlags` is a mandatory attribute which is a bitmap representing the following status flags:

Bits	Name	Description
0	In Alarm	Reserved - unused for Binary Input (Basic) cluster
1	Fault	• 1: Optional attribute `u8Reliability` is used and does not have a value of NO_FAULT_DETECTED • 0: Otherwise
2	Overridden	• 1: Cluster has been over-ridden by a local mechanism (`bPresentValue` and `u8Reliability` will not track input) • 0: Otherwise
3	Out Of Service	• 1: Attribute `bOutOfService` is set to TRUE • 0: Otherwise
4-7	-	Reserved

- `u32ApplicationType` is an optional attribute which is a bitmap representing the application type, as follows:

Bits	Field Name	Description
0-15	Index	Specific application usage (for example, Boiler Status). There is a set of possible usages for each value of Type (see below). For these lists, refer to the attribute description in the ZCL Specification.
16-23	Type	Application domain. For the Basic Input cluster, this can be set to 0x00 (HVAC) or 0x01 (Security).
24-31	Group	Identifier for the cluster that this attribute is part of (not the Clus-ter ID). For the Binary Input (Basic) cluster, this is 0x03.

```
u8AttributeReportingStatus is an optional attribute that should be enabled when attribute
u16ClusterRevision is a mandatory attribute that specifies the revision of the cluster spe
```

19.3.3 Attributes for Default Reporting

The following attributes of the Binary Input (Basic) cluster can be selected for default reporting:

```
bPresentValue
u8AttributeReportingStatus
```

Attribute reporting (including default reporting) is described in Appendix B. Enabling reports for these attributes is described in Appendix B.3.6.

19.3.4 Functions

The following Binary Input (Basic) cluster function is provided in the NXP implementation of the ZCL:

- **Function Page**
- eCLD_BinaryInputBasicCreateBinaryInputBasic 432

The cluster attributes can be accessed using the general attribute read/write functions, as described in Section 2.3.

19.3.4.1 eCLD_BinaryInputBasicCreateBinaryInputBasic

```
teZCL_Status eCLD_BinaryInputBasicCreateBinaryInputBasic(
        tsZCL_ClusterInstance *psClusterInstance,
        bool_t bIsServer,
        tsZCL_ClusterDefinition *psClusterDefinition,
        void *pvEndPointSharedStructPtr,
        uint8 *pu8AttributeControlBits);
```

Description

This function creates an instance of the Binary Input (Basic) 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 Binary Input (Basic) cluster instance on the endpoint, but instances of other clusters may also be created on the same endpoint by calling their corresponding creation functions.

Note: *This function must not be called for an endpoint on which a standard ZigBee device is used. In this case, the device and its supported clusters must be registered on the endpoint using the relevant device registration function.*

When used, this function must be called after the stack has been started and after the ZCL has been initialized.

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 Binary Input Basic cluster.

The function initializes the array elements to zero.

Parameters

psClusterInstance Pointer to structure containing information about the cluster instance to be created (see Section 6.1.16). This structure is updated by the function by initialising individual structure fields.

bIsServer Type of cluster instance (server or client) to be created:

TRUE - server

FALSE - client

psClusterDefinition Pointer to structure indicating the type of cluster to be created (see Section 6.1.2). In this case, this structure must contain the details of the Binary Input (Basic) cluster. This parameter can refer to a pre-filled structure called `sCLD_BinaryInputBasic` which is provided in the **BinaryInputBasic.h** file.

pvEndPointSharedStructPtr Pointer to the shared structure used for attribute storage. This parameter should be the address of the structure of type `tsCLD_BinaryInputBasic` which defines the attributes of Binary Input (Basic) cluster. The function initializes the attributes with default values.

pu8AttributeControlBits Pointer to an array of **uint8** values, with one element for each attribute in the cluster (see above).

Returns

E_ZCL_SUCCESS
E_ZCL_FAIL
E_ZCL_ERR_PARAMETER_NULL
E_ZCL_ERR_INVALID_VALUE

19.3.5 Enumerations

19.3.5.1 teCLD_BinaryInputBasicCluster_AttrID

The following structure contains the enumerations used to identify the attributes of the Binary Input (Basic) cluster.

```
typedef enum
{
    E_CLD_BINARY_INPUT_BASIC_ATTR_ID_ACTIVE_TEXT,
    E_CLD_BINARY_INPUT_BASIC_ATTR_ID_DESCRIPTION,
    E_CLD_BINARY_INPUT_BASIC_ATTR_ID_INACTIVE_TEXT,
    E_CLD_BINARY_INPUT_BASIC_ATTR_ID_OUT_OF_SERVICE,
    E_CLD_BINARY_INPUT_BASIC_ATTR_ID_POLARITY,
    E_CLD_BINARY_INPUT_BASIC_ATTR_ID_PRESENT_VALUE,
    E_CLD_BINARY_INPUT_BASIC_ATTR_ID_RELIABILITY,
    E_CLD_BINARY_INPUT_BASIC_ATTR_ID_STATUS_FLAGS,
    E_CLD_BINARY_INPUT_BASIC_ATTR_ID_APPLICATION_TYPE
} teCLD_BinaryInputBasicCluster_AttrID;
```

19.3.5.2 teCLD_BinaryInputBasic_Polarity

The following structure contains the enumerations used to specify the value of the `u8Polarity` attribute (see Section 19.3.2).

```
typedef enum
{
   E_CLD_ BINARY_INPUT_BASIC_POLARITY_NORMAL                              ,
   E_CLD_ BINARY_INPUT_BASIC_POLARITY_REVERSE
}teCLD_BinaryInputBasic_Polarity
```

19.3.5.3 teCLD_BinaryInputBasic_Reliability

The following structure contains the enumerations used to report the value of the `u8Reliability` attribute (see Section 19.3.2).

```
typedef enum
{
   E_CLD_ BINARY_INPUT_BASIC_RELIABILITY_NO_FAULT_DETECTED,
   E_CLD_ BINARY_INPUT_BASIC_RELIABILITY_NO_SENSOR,
   E_CLD_ BINARY_INPUT_BASIC_RELIABILITY_OVER_RANGE,
   E_CLD_ BINARY_INPUT_BASIC_RELIABILITY_UNDER_RANGE,
   E_CLD_ BINARY_INPUT_BASIC_RELIABILITY_OPEN_LOOP,
   E_CLD_ BINARY_INPUT_BASIC_RELIABILITY_SHORTED_LOOP,
```

```
    E_CLD_ BINARY_INPUT_BASIC_RELIABILITY_NO_OUTPUT,
    E_CLD_ BINARY_INPUT_BASIC_RELIABILITY_UNRELIABLE_OTHER,
    E_CLD_ BINARY_INPUT_BASIC_RELIABILITY_PROCESS_ERROR,
    E_CLD_ BINARY_INPUT_BASIC_RELIABILITY_CONFIGURATION_ERROR
}teCLD_BinaryInputBasic_Reliability;
```

19.3.6 Compile-time options

To enable the Binary Input (Basic) cluster in the code to be built, it is necessary to add the following to the **zcl_options.h** file:

```
#define CLD_BINARY_INPUT_BASIC
```

In addition, to include the software for a cluster client or server or both, it is necessary to add one or both of the following to the same file:

```
#define BINARY_INPUT_BASIC_CLIENT
#define BINARY_INPUT_BASIC_SERVER
```

Optional Attributes

The optional attributes for the Binary Input (Basic) cluster (see Section 19.3.2) are enabled by defining:

- CLD_BINARY_INPUT_BASIC_ATTR_ACTIVE_TEXT
- CLD_BINARY_INPUT_BASIC_ATTR_DESCRIPTION
- CLD_BINARY_INPUT_BASIC_ATTR_INACTIVE_TEXT
- CLD_BINARY_INPUT_BASIC_ATTR_POLARITY
- CLD_BINARY_INPUT_BASIC_ATTR_RELIABILITY
- CLD_BINARY_INPUT_BASIC_ATTR_APPLICATION_TYPE

Global Attributes

Add this line to enable the optional Attribute Reporting Status attribute:

```
#define CLD_BINARY_INPUT_BASIC_ATTR_ID_ATTRIBUTE_REPORTING_STATUS
```

Add this line to define the value (n) of the Cluster Revision attribute:

```
#define CLD_BINARY_INPUT_BASIC_CLUSTER_REVISION <n>
```

The default value is 1, which corresponds to the revision of the cluster in the ZCL r6 specification (see Section 2.4).

19.4 Binary Output (Basic)

This chapter describes the Binary Output (Basic) cluster, which provides an interface for setting the state of a binary (two-state) output.

The Binary Output (Basic) cluster has a Cluster ID of 0x0010.

19.4.1 Overview

The Binary Output (Basic) cluster provides an interface for setting the state of a binary output and its associated characteristics. It is typically used to implement a controller that produces a two-state output signal.

To use the functionality of this cluster, you must include the file **BinaryOutputBasic.h** in your application and enable the cluster by defining CLD_BINARY_OUTPUT_BASIC in the **zcl_options.h** file.

A Binary Output (Basic) cluster instance can act as either a client or a server. The inclusion of the client or server software must be pre-defined in the application's compile-time options (in addition, if the cluster is to reside on a custom endpoint then the role of client or server must also be specified when creating the cluster instance).

The compile-time options for the Binary Output (Basic) cluster are fully detailed in Section 19.4.6.

19.4.2 Binary Output (Basic) Structure and Attributes

The structure definition for the Binary Output (Basic) cluster is:

```
typedef struct
{
#ifdef BINARY_OUTPUT_BASIC_SERVER
#ifdef CLD_BINARY_OUTPUT_BASIC_ATTR_ACTIVE_TEXT
    tsZCL_CharacterString       sActiveText;
    uint8                       au8ActiveText[16];
#endif
#ifdef CLD_BINARY_OUTPUT_BASIC_ATTR_DESCRIPTION
    tsZCL_CharacterString       sDescription;
    uint8                       au8Description[16];
#endif
#ifdef CLD_BINARY_OUTPUT_BASIC_ATTR_INACTIVE_TEXT
    tsZCL_CharacterString       sInactiveText;
    uint8                       au8InactiveText[16];
#endif
#ifdef CLD_BINARY_OUTPUT_BASIC_ATTR_MINIMUM_OFF_TIME
    zuint32                     u32MinimumOffTime;
#endif
#ifdef CLD_BINARY_OUTPUT_BASIC_ATTR_MINIMUM_ON_TIME
    zuint32                     u32MinimumOnTime;
#endif
    zbool                       bOutOfService;
#ifdef CLD_BINARY_OUTPUT_BASIC_ATTR_POLARITY
    zenum8                      u8Polarity;
#endif
    zbool                       bPresentValue;
#ifdef CLD_BINARY_OUTPUT_BASIC_ATTR_RELIABILITY
    zenum8                      u8Reliability;
#endif
#ifdef CLD_BINARY_OUTPUT_BASIC_ATTR_RELINQUISH_DEFAULT
    zbool                       bRelinquishDefault;
#endif
    zbmap8                      u8StatusFlags;
#ifdef CLD_BINARY_OUTPUT_BASIC_ATTR_APPLICATION_TYPE
    zuint32                     u32ApplicationType;
#endif
#ifdef CLD_BINARY_OUTPUT_BASIC_ATTR_ATTRIBUTE_REPORTING_STATUS
    zenum8                      u8AttributeReportingStatus;
#endif
#endif
    zuint16                     u16ClusterRevision;
```

```
} tsCLD_BinaryOutputBasic;
```

- The following optional pair of attributes are used to store a human readable description of the active state of a binary output (e.g. "Open Window 3"):
  - `sActiveText` is a `tsZCL_CharacterString` structure (see Section 6.1.14) for a string of up to 16 characters representing the description
  - `au8ActiveText[16]` is a byte-array which contains the character data bytes representing the description

    If these attributes are used, the ‘Inactive Text’ attributes must also be used.
- The following optional pair of attributes are used to store a human readable description of the usage of the binary output (e.g. "Control Window 3"):
  - `sDescription` is a `tsZCL_CharacterString` structure (see Section 6.1.14) for a string of up to 16 characters representing the description
  - `au8Description[16]` is a byte-array which contains the character data bytes representing the description
- The following optional pair of attributes are used to store a human readable description of the inactive state of a binary output (e.g. "Close Window 3"):
  - `sInactiveText` is a `tsZCL_CharacterString` structure (see Section 6.1.14) for a string of up to 16 characters representing the description
  - `au8InactiveText[16]` is a byte-array which contains the character data bytes representing the description

    If these attributes are used, the ‘Active Text’ attributes must also be used.

`u32MinimumOffTime` is an optional attribute which represents the minimum time, in seconds, for which the binary output will remain in the inactive state (0).

- `u32MinimumOnTime` is an optional attribute which represents the minimum time, in seconds, for which the binary output will remain in the active state (1).
- `bOutOfService` is a mandatory attribute which indicates whether the binary output is currently in or out of service:
  - TRUE: Out of service
  - FALSE: In service

    If this attribute is set to TRUE, the `bPresentValue` attribute will not be used to control the binary output.
- `u8Polarity` is an optional attribute which indicates the relationship between the value of the `bPresentValue` attribute and the physical state of the output:
  - E_CLD_ BINARY_OUTPUT_BASIC_POLARITY_NORMAL (0x00): The active (1) state of `bPresentValue` corresponds to the active/on state of the physical output
  - E_CLD_ BINARY_OUTPUT_BASIC_POLARITY_REVERSE (0x01): The active (1) state of `bPresentValue` corresponds to the inactive/off state of the physical output
- `bPresentValue` is a mandatory attribute representing the current state of the binary output (this attribute is updated by the application):
  - TRUE: Output is in the ‘active’ state
  - FALSE: Output is in the ‘inactive’ state

    The interpretation `bPresentValue` in relation to the physical state of the output is determined by the setting of the `u8Polarity` attribute.
- `u8Reliability` is an optional attribute which indicates whether the value contained in `bPresentValue` is reliable or why it might be unreliable:
  - E_CLD_BINARY_OUTPUT_BASIC_RELIABILITY_NO_FAULT_DETECTED
  - E_CLD_BINARY_OUTPUT_BASIC_RELIABILITY_OVER_RANGE
  - E_CLD_BINARY_OUTPUT_BASIC_RELIABILITY_UNDER_RANGE
  - E_CLD_BINARY_OUTPUT_BASIC_RELIABILITY_OPEN_LOOP

- E_CLD_BINARY_OUTPUT_BASIC_RELIABILITY_SHORTED_LOOP
- E_CLD_BINARY_OUTPUT_BASIC_RELIABILITY_UNRELIABLE_OTHER
- E_CLD_BINARY_OUTPUT_BASIC_RELIABILITY_PROCESS_ERROR
- E_CLD_BINARY_OUTPUT_BASIC_RELIABILITY_CONFIGURATION_ERROR

`fRelinquishDefault` is an optional attribute representing the default value to be used for `bPresentValue` when the supplied value is invalid.

- `u8StatusFlags` is a mandatory attribute which is a bitmap representing the following status flags:

Bits	Name	Description
0	In Alarm	Reserved - unused for Binary Output (Basic) cluster
1	Fault	• 1: Optional attribute `u8Reliability` is used and does not have a value of NO_FAULT_DETECTED • 0: Otherwise
2	Overridden	• 1: Cluster has been over-ridden by a local mechanism (`bPresentValue` and `u8Reliability` will not track input) • 0: Otherwise
3	Out Of Service	• 1: Attribute `bOutOfService` is set to TRUE • 0: Otherwise
4-7	-	Reserved

- `u32ApplicationType` is an optional attribute which is a bitmap representing the application type, as follows:

Bits	Field Name	Description
0-15	Index	Specific application usage (e.g. Heating Valve). There is a set of possible usages for each value of Type (see below). For these lists, refer to the attribute description in the ZCL Specification.
16-23	Type	Application domain. For the Basic Output cluster, this can be set to 0x00 (HVAC) or 0x01 (Security).
24-31	Group	Identifier for the cluster that this attribute is part of (not the Clus-ter ID). For the Binary Output (Basic) cluster, this is 0x04.

```
u8AttributeReportingStatus is an optional attribute that should be enabled when attribute
u16ClusterRevision is a mandatory attribute that specifies the revision of the cluster spe
```

19.4.3 Attributes for Default Reporting

The following attributes of the Binary Output (Basic) cluster can be selected for default reporting:

```
bPresentValue
u8AttributeReportingStatus
```

Attribute reporting (including default reporting) is described in Appendix B. Enabling reports for these attributes is described in Appendix B.3.6.

19.4.4 Functions

The following Binary Output (Basic) cluster function is provided in the NXP implementation of the ZCL:

The cluster attributes can be accessed using the general attribute read/write functions, as described in Section 2.3.

19.4.4.1 eCLD_BinaryOutputBasicCreateBinaryOutputBasic

```
teZCL_Status eCLD_BinaryOutputBasicCreateBinaryOutputBasic(
        tsZCL_ClusterInstance *psClusterInstance,
        bool_t bIsServer,
        tsZCL_ClusterDefinition *psClusterDefinition,
        void *pvEndPointSharedStructPtr,
        uint8 *pu8AttributeControlBits);
```

Description

This function creates an instance of the Binary Output (Basic) 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 Binary Output (Basic) cluster instance on the endpoint, but instances of other clusters may also be created on the same endpoint by calling their corresponding creation functions.

Note: *This function must not be called for an endpoint on which a standard ZigBee device is used. In this case, the device and its supported clusters must be registered on the endpoint using the relevant device registration function.*

When used, this function must be called after the stack has been started and after the ZCL has been initialized.

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 Binary Output Basic cluster.

The function initializes the array elements to zero.

Parameters

psClusterInstance Pointer to structure containing information about the cluster instance to be created (see Section 6.1.16). This structure is updated by the function by initialising individual structure fields.

bIsServer Type of cluster instance (server or client) to be created:

TRUE - server

FALSE - client

psClusterDefinition Pointer to structure indicating the type of cluster to be created (see Section 6.1.2). In this case, this structure must contain the details of the Binary Output (Basic) cluster. This parameter can refer to a pre-filled structure called `sCLD_BinaryOutputBasic` which is provided in the **BinaryOutputBasic.h** file.

pvEndPointSharedStructPtr Pointer to the shared structure used for attribute storage. This parameter should be the address of the structure of type `tsCLD_BinaryOutputBasic` which defines the attributes of Binary Output (Basic) cluster. The function initializes the attributes with default values.

pu8AttributeControlBits Pointer to an array of **uint8** values, with one element for each attribute in the cluster (see above).

Returns

E_ZCL_SUCCESS
E_ZCL_FAIL
E_ZCL_ERR_PARAMETER_NULL
E_ZCL_ERR_INVALID_VALUE

19.4.5 Enumerations

19.4.5.1 teCLD_BinaryOutputBasicCluster_AttrID

The following structure contains the enumerations used to identify the attributes of the Binary Output (Basic) cluster.

```
typedef enum
{
    E_CLD_BINARY_OUTPUT_BASIC_ATTR_ID_ACTIVE_TEXT,
    E_CLD_BINARY_OUTPUT_BASIC_ATTR_ID_DESCRIPTION,
    E_CLD_BINARY_OUTPUT_BASIC_ATTR_ID_INACTIVE_TEXT,
    E_CLD_BINARY_OUTPUT_BASIC_ATTR_ID_MINIMUM_OFF_TIME,
    E_CLD_BINARY_OUTPUT_BASIC_ATTR_ID_MINIMUM_ON_TIME,
    E_CLD_BINARY_OUTPUT_BASIC_ATTR_ID_OUT_OF_SERVICE,
    E_CLD_BINARY_OUTPUT_BASIC_ATTR_ID_POLARITY,
    E_CLD_BINARY_OUTPUT_BASIC_ATTR_ID_PRESENT_VALUE,
    E_CLD_BINARY_OUTPUT_BASIC_ATTR_ID_RELIABILITY,
    E_CLD_BINARY_OUTPUT_BASIC_ATTR_ID_RELINQUISH_DEFAULT,
    E_CLD_BINARY_OUTPUT_BASIC_ATTR_ID_STATUS_FLAGS,
    E_CLD_BINARY_OUTPUT_BASIC_ATTR_ID_APPLICATION_TYPE,
} teCLD_BinaryOutputBasicCluster_AttrID;
```

19.4.5.2 teCLD_BinaryOutputBasic_Polarity

The following structure contains the enumerations used to specify the value of the `u8Polarity` attribute (see Section 19.4.2).

```
typedef enum
{
    E_CLD_BINARY_OUTPUT_BASIC_POLARITY_NORMAL,
    E_CLD_BINARY_OUTPUT_BASIC_POLARITY_REVERSE
}teCLD_BinaryOutputBasic_Polarity
```

19.4.5.3 teCLD_BinaryOutputBasic_Reliability

The following structure contains the enumerations used to report the value of the `u8Reliability` attribute (see Section 19.4.2).

```
typedef enum
{
    E_CLD_BINARY_OUTPUT_BASIC_RELIABILITY_NO_FAULT_DETECTED,
    E_CLD_BINARY_OUTPUT_BASIC_RELIABILITY_OVER_RANGE,
    E_CLD_BINARY_OUTPUT_BASIC_RELIABILITY_UNDER_RANGE,
    E_CLD_BINARY_OUTPUT_BASIC_RELIABILITY_OPEN_LOOP,
    E_CLD_BINARY_OUTPUT_BASIC_RELIABILITY_SHORTED_LOOP,
    E_CLD_BINARY_OUTPUT_BASIC_RELIABILITY_UNRELIABLE_OTHER,
    E_CLD_BINARY_OUTPUT_BASIC_RELIABILITY_PROCESS_ERROR,
```

```
    E_CLD_BINARY_OUTPUT_BASIC_RELIABILITY_CONFIGURATION_ERROR
 }teCLD_BinaryOutputBasic_Reliability;
```

19.4.6 Compile-time options

To enable the Binary Output (Basic) cluster in the code to be built, it is necessary to add the following to the **zcl_options.h** file:

```
#define CLD_BINARY_OUTPUT_BASIC
```

In addition, to include the software for a cluster client or server or both, it is necessary to add one or both of the following to the same file:

```
#define BINARY_OUTPUT_BASIC_CLIENT
#define BINARY_OUTPUT_BASIC_SERVER
```

Optional Attributes

The optional attributes for the Binary Output (Basic) cluster (see Section 19.4.2) are enabled by defining:

- CLD_BINARY_OUTPUT_BASIC_ATTR_ACTIVE_TEXT
- CLD_BINARY_OUTPUT_BASIC_ATTR_DESCRIPTION
- CLD_BINARY_OUTPUT_BASIC_ATTR_INACTIVE_TEXT
- CLD_BINARY_OUTPUT_BASIC_ATTR_MINIMUM_OFF_TIME
- CLD_BINARY_OUTPUT_BASIC_ATTR_MINIMUM_ON_TIME
- CLD_BINARY_OUTPUT_BASIC_ATTR_POLARITY
- CLD_BINARY_OUTPUT_BASIC_ATTR_RELIABILITY
- CLD_BINARY_OUTPUT_BASIC_ATTR_RELINQUISH_DEFAULT
- CLD_BINARY_OUTPUT_BASIC_ATTR_APPLICATION_TYPE
- CLD_BINARY_OUTPUT_BASIC_ATTR_ATTRIBUTE_REPORTING_STATUS

19.5 Multistate Input (Basic)

This chapter describes the Multistate Input (Basic) cluster, which provides an interface for accessing a multistate measurement (that can take one of a set of fixed states).

The Multistate Input (Basic) cluster has a Cluster ID of 0x0012.

19.5.1 Overview

The Multistate Input (Basic) cluster provides an interface for accessing a multistate measurement and its associated characteristics. It is typically used in a sensor that measures a physical quantity that can take one of a discrete number of states.

To use the functionality of this cluster, you must include the file **MultistateInputBasic.h** in your application and enable the cluster by defining CLD_MULTISTATE_INPUT_BASIC in the **zcl_options.h** file.

A Multistate Input (Basic) cluster instance can act as either a client or a server. The inclusion of the client or server software must be pre-defined in the application's compile-time options (in addition, if the cluster is to reside on a custom endpoint then the role of client or server must also be specified when creating the cluster instance).

The compile-time options for the Multistate Input (Basic) cluster are fully detailed in Section 19.5.6.

19.5.2 Multistate Input (Basic) Structure and Attributes

The structure definition for the Multistate Input (Basic) cluster is:

```
typedef struct
{
#ifdef MULTISTATE_INPUT_BASIC_SERVER
#ifdef CLD_MULTISTATE_INPUT_BASIC_ATTR_DESCRIPTION
    tsZCL_CharacterString       sDescription;
    uint8                       au8Description[16];
#endif
    zuint16                     u16NumberOfStates;
    zbool                       bOutOfService;
    zuint16                     u16PresentValue;
#ifdef CLD_MULTISTATE_INPUT_BASIC_ATTR_RELIABILITY
    zenum8                      u8Reliability;
#endif
    zbmap8                      u8StatusFlags;
#ifdef CLD_MULTISTATE_INPUT_BASIC_ATTR_APPLICATION_TYPE
    zuint32                     u32ApplicationType;
#endif
#ifdef CLD_MULTISTATE_INPUT_BASIC_ATTR_ATTRIBUTE_REPORTING_STATUS
    zenum8                      u8AttributeReportingStatus;
#endif
#endif
    zuint16                     u16ClusterRevision;
} tsCLD_MultistateInputBasic;
```

- The following optional pair of attributes are used to store a human readable description of the usage of the multistate input (e.g. "Alarm Status"):
  - `sDescription` is a `tsZCL_CharacterString` structure (see Section 6.1.14) for a string of up to 16 characters representing the description
  - `au8Description[16]` is a byte-array which contains the character data bytes representing the description

`u16NumberOfStates` is a mandatory attribute which indicates the number of discrete states that the input can take.

- `bOutOfService` is a mandatory attribute which indicates whether the multistate input is currently in or out of service:
  - TRUE: Out of service
  - FALSE In service

    If this attribute is set to TRUE, the `u16PresentValue` attribute is not updated to contain the current state of the input.
- `u16PresentValue` is a mandatory attribute representing the latest state of the input (this attribute is updated when the multistate input changes).

  By default this attribute is read-only, but it becomes readable and writable when `bOutOfService` is set to TRUE.
- `u8Reliability` is an optional attribute which indicates whether the value reported through `fPresentValue` is reliable or why it might be unreliable:
  - E_CLD_MULTISTATE_INPUT_BASIC_RELIABILITY_NO_FAULT_DETECTED
  - E_CLD_MULTISTATE_INPUT_BASIC_RELIABILITY_NO_SENSOR
  - E_CLD_MULTISTATE_INPUT_BASIC_RELIABILITY_OVER_RANGE
  - E_CLD_MULTISTATE_INPUT_BASIC_RELIABILITY_UNDER_RANGE
  - E_CLD_MULTISTATE_INPUT_BASIC_RELIABILITY_OPEN_LOOP

  - E_CLD_MULTISTATE_INPUT_BASIC_RELIABILITY_SHORTED_LOOP
  - E_CLD_MULTISTATE_INPUT_BASIC_RELIABILITY_NO_OUTPUT
  - E_CLD_MULTISTATE_INPUT_BASIC_RELIABILITY_UNRELIABLE_OTHER
  - E_CLD_MULTISTATE_INPUT_BASIC_RELIABILITY_PROCESS_ERROR
  - E_CLD_MULTISTATE_INPUT_BASIC_RELIABILITY_MULTISTATE_FAULT
- `u8StatusFlags` is a mandatory attribute which is a bitmap representing the following status flags:

Bits	Name	Description
0	In Alarm	Reserved - unused for Multistate Input (Basic) cluster
1	Fault	• 1: Optional attribute `u8Reliability` is used and does not have a value of NO_FAULT_DETECTED • 0: Otherwise
2	Overridden	• 1: Cluster has been over-ridden by a local mechanism (`fPresentValue` and `u8Reliability` will not track input) • 0: Otherwise
3	Out Of Service	• 1: Attribute `bOutOfService` is set to TRUE • 0: Otherwise
4-7	-	Reserved

- `u32ApplicationType` is an optional attribute that is a bitmap representing the application type, as follows:

Bits	Field Name	Description
0-15	Index	Specific application usage in terms of the states supported (e.g. Off/On/Auto). For the list of usages, refer to the attribute description in the ZCL Specification.
16-23	Type	Application domain. For the Multistate Input cluster, this can only be is set to 0x00 (HVAC).
24-31	Group	Identifier for the cluster that this attribute is part of (not the Clus-ter ID). For the Multistate Input cluster, this is 0x0D.

- `u8AttributeReportingStatus` is an optional attribute that should be enabled when attribute reporting is used for the cluster (see Section 2.3.5). The value of this attribute indicates whether there are attribute reports still pending (0x00) or the attribute reports are complete (0x01) - all other values are reserved. This attribute is also described in Section 2.4.
- `u16ClusterRevision` is a mandatory attribute that specifies the revision of the cluster specification on which this cluster instance is based. The cluster specification in the ZCL r6 corresponds to a cluster revision of `1`. The value is incremented by one for each subsequent revision of the cluster specification. This attribute is also described in Section 2.4.

19.5.3 Attributes for Default Reporting

The following attributes of the Multistate Input (Basic) cluster can be selected for default reporting:

```
u16PresentValue
u8AttributeReportingStatus
```

Attribute reporting (including default reporting) is described in Appendix B. Enabling reports for these attributes is described in Appendix B.3.6.

19.5.4 Functions

The following Multistate Input (Basic) cluster function is provided in the NXP implementation of the ZCL:

-
- eCLD_MultistateInputBasicCreateMultistateInputBasic 450

The cluster attributes can be accessed using the general attribute read/write functions, as described in Section 2.3.

19.5.4.1 eCLD_MultistateInputBasicCreateMultistateInputBasic

```
teZCL_Status eCLD_MultistateInputBasicCreateMultistateInputBasic(
        tsZCL_ClusterInstance *psClusterInstance,
        bool_t bIsServer,
        tsZCL_ClusterDefinition *psClusterDefinition,
        void *pvEndPointSharedStructPtr,
        uint8 *pu8AttributeControlBits);
```

Description

This function creates an instance of the Multistate Input (Basic) 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 Multistate Input (Basic) cluster instance on the endpoint, but instances of other clusters may also be created on the same endpoint by calling their corresponding creation functions.

Note: *This function must not be called for an endpoint on which a standard ZigBee device is used. In this case, the device and its supported clusters must be registered on the endpoint using the relevant device registration function.*

When used, this function must be called after the stack has been started and after the ZCL has been initialized.

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 Multistate Input Basic cluster.

The function initializes the array elements to zero.

Parameters

psClusterInstance: Pointer to structure containing information about the cluster instance to be created (see Section 6.1.16). This structure is updated by the function by initialising individual structure fields.
bIsServer: Type of cluster instance (server or client) to be created:
TRUE - server
FALSE - client
psClusterDefinition: Pointer to structure indicating the type of cluster to be created (see Section 6.1.2). In this case, this structure must contain the details of the Multistate Input (Basic) cluster. This parameter can refer to a pre-filled structure called `sCLD_MultistateInputBasic` which is provided in the **MultistateInputBasic.h** file.
pvEndPointSharedStructPtr: Pointer to the shared structure used for attribute storage. This parameter should be the address of the structure of type `tsCLD_MultistateInputBasic` which defines the attributes of Multistate Input (Basic) cluster. The function initializes the attributes with default values.

pu8AttributeControlBits: Pointer to an array of **uint8** values, with one element for each attribute in the cluster (see above).

Returns

E_ZCL_SUCCESS
E_ZCL_FAIL
E_ZCL_ERR_PARAMETER_NULL
E_ZCL_ERR_INVALID_VALUE

19.5.5 Enumerations

19.5.5.1 teCLD_MultistateInputBasicCluster_AttrID

The following structure contains the enumerations used to identify the attributes of the Multistate Input (Basic) cluster.

```
typedef enum
{
     E_CLD_MULTISTATE_INPUT_BASIC_ATTR_ID_DESCRIPTION,
     E_CLD_MULTISTATE_INPUT_BASIC_ATTR_ID_NUMBER_OF_STATES,
     E_CLD_MULTISTATE_INPUT_BASIC_ATTR_ID_OUT_OF_SERVICE,
     E_CLD_MULTISTATE_INPUT_BASIC_ATTR_ID_PRESENT_VALUE,
     E_CLD_MULTISTATE_INPUT_BASIC_ATTR_ID_RELIABILITY,
     E_CLD_MULTISTATE_INPUT_BASIC_ATTR_ID_STATUS_FLAGS,
     E_CLD_MULTISTATE_INPUT_BASIC_ATTR_ID_APPLICATION_TYPE,
}teCLD_MultistateInputBasicCluster_AttrID;
```

19.5.5.2 teCLD_MultistateInputBasic_Reliability

The following structure contains the enumerations used to report the value of the `u8Reliability` attribute (see Section 19.5.2).

```
typedef enum
{
   E_CLD_MULTISTATE_INPUT_BASIC_RELIABILITY_NO_FAULT_DETECTED,
   E_CLD_MULTISTATE_INPUT_BASIC_RELIABILITY_NO_SENSOR,
   E_CLD_MULTISTATE_INPUT_BASIC_RELIABILITY_OVER_RANGE,
   E_CLD_MULTISTATE_INPUT_BASIC_RELIABILITY_UNDER_RANGE,
   E_CLD_MULTISTATE_INPUT_BASIC_RELIABILITY_OPEN_LOOP,
   E_CLD_MULTISTATE_INPUT_BASIC_RELIABILITY_SHORTED_LOOP,
   E_CLD_MULTISTATE_INPUT_BASIC_RELIABILITY_NO_OUTPUT,
   E_CLD_MULTISTATE_INPUT_BASIC_RELIABILITY_UNRELIABLE_OTHER,
   E_CLD_MULTISTATE_INPUT_BASIC_RELIABILITY_PROCESS_ERROR,
   E_CLD_MULTISTATE_INPUT_BASIC_RELIABILITY_MULTISTATE_FAULT,
   E_CLD_MULTISTATE_INPUT_BASIC_RELIABILITY_CONFIGURATION_ERROR
}teCLD_MultistateInputBasic_Reliability;
```

19.5.6 Compile-time options

To enable the Multistate Input (Basic) cluster in the code to be built, it is necessary to add the following to the **zcl_options.h** file:

```
#define CLD_MULTISTATE_INPUT_BASIC
```

In addition, to include the software for a cluster client or server or both, it is necessary to add one or both of the following to the same file:

```
#define MULTISTATE_INPUT_BASIC_CLIENT
#define MULTISTATE_INPUT_BASIC_SERVER
```

Optional Attributes

The optional attributes for the Multistate Input (Basic) cluster (see Section 19.5.2) are enabled by defining:

- CLD_MULTISTATE_INPUT_BASIC_ATTR_DESCRIPTION
- CLD_MULTISTATE_INPUT_BASIC_ATTR_RELIABILITY
- CLD_MULTISTATE_INPUT_BASIC_ATTR_APPLICATION_TYPE
- CLD_MULTISTATE_INPUT_BASIC_ATTR_ATTRIBUTE_REPORTING_STATUS

Global Attributes

Add this line to enable the optional Attribute Reporting Status attribute:

```
#define CLD_MULTISTATE_INPUT_BASIC_ATTR_ID_ATTRIBUTE_REPORTING_STATUS
```

Add this line to define the value (n) of the Cluster Revision attribute:

```
#define CLD_MULTISTATE_INPUT_BASIC_CLUSTER_REVISION <n>
```

The default value is `1`, which corresponds to the revision of the cluster in the ZCL r6 specification (see Section 2.4).

19.6 Multistate Output (Basic)

This chapter describes the Multistate Output (Basic) cluster, which provides an interface for setting the value of a multistate output (that can take one of a set of fixed states).

The Multistate Output (Basic) cluster has a Cluster ID of 0x0013.

19.6.1 Overview

The Multistate Input (Basic) cluster provides an interface for setting the value of a multistate output and its associated characteristics. It is typically used in a controller which outputs a control signal that can be set to one of a discrete number of states.

To use the functionality of this cluster, you must include the file **MultistateOutputBasic.h** in your application and enable the cluster by defining CLD_MULTISTATE_OUTPUT_BASIC in the **zcl_options.h** file.

An Multistate Output (Basic) cluster instance can act as either a client or a server. The inclusion of the client or server software must be pre-defined in the application's compile-time options (in addition, if the cluster is to reside on a custom endpoint then the role of client or server must also be specified when creating the cluster instance).

The compile-time options for the Multistate Output (Basic) cluster are fully detailed in Section 19.6.6.

19.6.2 Multistate Output (Basic) Structure and Attributes

The structure definition for the Multistate Output (Basic) cluster is:

```
typedef struct
{
#ifdef MULTISTATE_OUTPUT_BASIC_SERVER
#ifdef CLD_MULTISTATE_OUTPUT_BASIC_ATTR_DESCRIPTION
    tsZCL_CharacterString       sDescription;
    uint8                      au8Description[16];
#endif
    zuint16                     u16NumberOfStates;
    zbool                      bOutOfService;
    zuint16                     u16PresentValue;
#ifdef CLD_MULTISTATE_OUTPUT_BASIC_ATTR_RELIABILITY
    zenum8                     u8Reliability;
#endif
#ifdef CLD_MULTISTATE_OUTPUT_BASIC_ATTR_RELINQUISH_DEFAULT
    zuint16                   u16RelinquishDefault;
#endif
    zbmap8                     u8StatusFlags;
#ifdef CLD_MULTISTATE_OUTPUT_BASIC_ATTR_APPLICATION_TYPE
    zuint32                     u32ApplicationType;
#endif
#ifdef CLD_MULTISTATE_OUTPUT_BASIC_ATTR_ATTRIBUTE_REPORTING_STATUS
    zenum8                     u8AttributeReportingStatus;
#endif
#endif
    zuint16                u16ClusterRevision;
} tsCLD_MultistateOutputBasic;
```

- The following optional pairs of attributes are used to store a human readable description of the usage of the multistate output (for example "Alarm State"):
  - `sDescription` is a `tsZCL_CharacterString` structure (see Section 6.1.14) for a string of up to 16 characters representing the description
  - `au8Description[16]` is a byte-array which contains the character data bytes representing the description.
- `u16NumberOfStates` is a mandatory attribute that indicates the number of discrete states that the output can take.
- `bOutOfService` is a mandatory attribute that indicates whether the multistate output is in or out of service currently:
  - TRUE: Out of service
  - FALSE: In service

    If this attribute is set to TRUE, the `u16PresentValue` attribute is not used to control the multistate output.
- `u16PresentValue` is a mandatory attribute representing the latest multistate output value (this attribute is used to control the physical output).
- `u8Reliability` is an optional attribute that indicates whether the value contained in `u16PresentValue` is reliable or why it might be unreliable:
  - `E_CLD_MULTISTATE_OUTPUT_BASIC_RELIABILITY_NO_FAULT_DETECTED`
  - `E_CLD_MULTISTATE_OUTPUT_BASIC_RELIABILITY_OVER_RANGE`
  - `E_CLD_MULTISTATE_OUTPUT_BASIC_RELIABILITY_UNDER_RANGE`
  - `E_CLD_MULTISTATE_OUTPUT_BASIC_RELIABILITY_OPEN_LOOP`
  - `E_CLD_MULTISTATE_OUTPUT_BASIC_RELIABILITY_SHORTED_LOOP`
  - `E_CLD_MULTISTATE_OUTPUT_BASIC_RELIABILITY_UNRELIABLE_OTHER`
  - `E_CLD_MULTISTATE_OUTPUT_BASIC_RELIABILITY_PROCESS_ERROR`
  - `E_CLD_MULTISTATE_OUTPUT_BASIC_RELIABILITY_MULTISTATE_FAULT`
  - `E_CLD_MULTISTATE_OUTPUT_BASIC_RELIABILITY_CONFIGURATION_ERROR`

- `fRelinquishDefault` is an optional attribute representing the default value to be used for `u16PresentValue` when the supplied value is invalid.
- `u8StatusFlags` is a mandatory attribute, which is a bitmap representing the following status flags:

Bits	Name	Description
0	In Alarm	Reserved - unused for Multistate Output (Basic) cluster
1	Fault	• 1: Optional attribute `u8Reliability` is used and does not have a value of `NO_FAULT_DETECTED` • 0: Otherwise
2	Overridden	• 1: Cluster has been over-ridden by a local mechanism (`u16PresentValue` and `u8Reliability` do not track input) • 0: Otherwise
3	Out Of Service	• 1: Attribute `bOutOfService` is set to TRUE • 0: Otherwise
4-7	-	Reserved

- `u32ApplicationType` is an optional attribute which is a bitmap representing the application type, as follows:

Bits	Field Name	Description
0-15	Index	Specific application usage in terms of the states supported (for example, Off/On/Auto). For the list of usages, refer to the attribute description in the ZCL Specification.
16-23	Type	Application domain. For the Multistate Output cluster, this field can only be set to 0x00 (HVAC).
24-31	Group	Identifier for the cluster that this attribute is part of (not the Cluster ID). For the Multistate Output cluster, this is 0x0E.

- `u8AttributeReportingStatus` is an optional attribute that should be enabled when attribute reporting is used for the cluster (see Section 2.3.5). The value of this attribute indicates whether there are attribute reports still pending (0x00) or the attribute reports are complete (0x01) - all other values are reserved. This attribute is also described in Section 2.4.
- `u16ClusterRevision` is a mandatory attribute that specifies the revision of the cluster specification on which this cluster instance is based. The cluster specification in the ZCL r6 corresponds to a cluster revision of 1. The value is incremented by one for each subsequent revision of the cluster specification. This attribute is also described in Section 2.4.

19.6.3 Attributes for Default Reporting

The following attributes of the Multistate Output (Basic) cluster can be selected for default reporting:

```
u16PresentValue
u8AttributeReportingStatus
```

Attribute reporting (including default reporting) is described in Appendix B. Enabling reports for these attributes is described in Appendix B.3.6.

19.6.4 Functions

The following Multistate Output (Basic) cluster function is provided in the NXP implementation of the ZCL:

- eCLD_MultistateOutputBasicCreateMultistateOutputBasic

The cluster attributes can be accessed using the general attribute read/write functions, as described in Section 2.3.

19.6.4.1 eCLD_MultistateOutputBasicCreateMultistateOutputBasic

```
teZCL_Status eCLD_MultistateOutputBasicCreateMultistateOutputBasic(
       tsZCL_ClusterInstance *psClusterInstance,
       bool_t bIsServer,
       tsZCL_ClusterDefinition *psClusterDefinition,
       void *pvEndPointSharedStructPtr,
       uint8 *pu8AttributeControlBits);
```

Description

This function creates an instance of the Multistate Output (Basic) 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 an Multistate Output (Basic) cluster instance on the endpoint, but instances of other clusters may also be created on the same endpoint by calling their corresponding creation functions.

Note: *This function must not be called for an endpoint on which a standard ZigBee device is used. In this case, the device and its supported clusters must be registered on the endpoint using the relevant device registration function.*

When used, this function must be called after the stack has been started and after the ZCL has been initialized.

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 Multistate Output Basic cluster.

The function initializes the array elements to zero.

Parameters

psClusterInstance Pointer to structure containing information about the cluster instance to be created (see Section 6.1.16). This structure is updated by the function by initialising individual structure fields.
bIsServer Type of cluster instance (server or client) to be created:
TRUE - server
FALSE - client
psClusterDefinition: Pointer to structure indicating the type of cluster to be created (see Section 6.1.2). In this case, this structure must contain the details of the Multistate Output (Basic) cluster. This parameter can refer to a pre-filled structure called `sCLD_MultistateOutputBasic` which is provided in the **MultistateOutputBasic.h** file.
pvEndPointSharedStructPtr: Pointer to the shared structure used for attribute storage. This parameter should be the address of the structure of type `tsCLD_MultistateOutputBasic` which defines the attributes of Multistate Output (Basic) cluster. The function initializes the attributes with default values.
pu8AttributeControlBits: Pointer to an array of **uint8** values, with one element for each attribute in the cluster (see above).

Returns

E_ZCL_SUCCESS
E_ZCL_FAIL
E_ZCL_ERR_PARAMETER_NULL
E_ZCL_ERR_INVALID_VALUE

19.6.5 Enumerations

19.6.5.1 teCLD_MultistateOutputBasicCluster_AttrID

The following structure contains the enumerations used to identify the attributes of the Multistate Output (Basic) cluster.

```
typedef enum {
   E_CLD_MULTISTATE_OUTPUT_BASIC_ATTR_ID_DESCRIPTION,
   E_CLD_MULTISTATE_OUTPUT_BASIC_ATTR_ID_NUMBER_OF_STATES,
   E_CLD_MULTISTATE_OUTPUT_BASIC_ATTR_ID_OUT_OF_SERVICE,
   E_CLD_MULTISTATE_OUTPUT_BASIC_ATTR_ID_PRESENT_VALUE,
   E_CLD_MULTISTATE_OUTPUT_BASIC_ATTR_ID_RELIABILITY,
   E_CLD_MULTISTATE_OUTPUT_BASIC_ATTR_ID_RELINQUISH_DEFAULT,
   E_CLD_MULTISTATE_OUTPUT_BASIC_ATTR_ID_STATUS_FLAGS,
   E_CLD_MULTISTATE_OUTPUT_BASIC_ATTR_ID_APPLICATION_TYPE,
} teCLD_MultistateOutputBasicCluster_AttrID;
```

19.6.5.2 teCLD_MultistateOutputBasic_Reliability

The following structure contains the enumerations used to report the value of the `u8Reliability` attribute (see Section 19.6.2).

```
typedef enum
{
   E_CLD_MULTISTATE_OUTPUT_BASIC_RELIABILITY_NO_FAULT_DETECTED,
   E_CLD_MULTISTATE_OUTPUT_BASIC_RELIABILITY_OVER_RANGE,
   E_CLD_MULTISTATE_OUTPUT_BASIC_RELIABILITY_UNDER_RANGE,
   E_CLD_MULTISTATE_OUTPUT_BASIC_RELIABILITY_OPEN_LOOP,
   E_CLD_MULTISTATE_OUTPUT_BASIC_RELIABILITY_SHORTED_LOOP,
   E_CLD_MULTISTATE_OUTPUT_BASIC_RELIABILITY_UNRELIABLE_OTHER,
   E_CLD_MULTISTATE_OUTPUT_BASIC_RELIABILITY_PROCESS_ERROR,
   E_CLD_MULTISTATE_OUTPUT_BASIC_RELIABILITY_MULTISTATE_FAULT,
   E_CLD_MULTISTATE_OUTPUT_BASIC_RELIABILITY_CONFIGURATION_ERROR
}teCLD_MultistateOutputBasic_Reliability;
```

19.6.6 Compile-time options

To enable the Multistate Output (Basic) cluster in the code to be built, it is necessary to add the following to the **zcl_options.h** file:

```
#define CLD_MULTISTATE_OUTPUT_BASIC
```

In addition, to include the software for a cluster client or server or both, it is necessary to add one or both of the following to the same file:

```
#define MULTISTATE_OUTPUT_BASIC_CLIENT
```

```
#define MULTISTATE_OUTPUT_BASIC_SERVER
```

Optional attributes

The optional attributes for the Multistate Output (Basic) cluster (see Section 19.6.2) are enabled by defining:

- CLD_MULTISTATE_OUTPUT_BASIC_ATTR_DESCRIPTION
- CLD_MULTISTATE_OUTPUT_BASIC_ATTR_RELIABILITY
- CLD_MULTISTATE_OUTPUT_BASIC_ATTR_RELINQUISH_DEFAULT
- CLD_MULTISTATE_OUTPUT_BASIC_ATTR_APPLICATION_TYPE
- CLD_MULTISTATE_OUTPUT_BASIC_ATTR_ATTRIBUTE_REPORTING_STATUS

Global attributes

Add this line to enable the optional Attribute Reporting Status attribute:

```
#define CLD_MULTISTATE_OUTPUT_BASIC_ATTR_ID_ATTRIBUTE_REPORTING_STATUS
```

Add this line to define the value (n) of the Cluster Revision attribute:

```
#define CLD_MULTISTATE_OUTPUT_BASIC_CLUSTER_REVISION <n>
```

The default value is 1, which corresponds to the revision of the cluster in the ZCL r6 specification (see Section 2.4).

20 Poll Control Cluster

This chapter describes the Poll Control cluster which provides an interface for remotely controlling the rate at which a ZigBee End Device polls its parent for data.

The Poll Control cluster has a Cluster ID of 0x0020.

20.1 Overview

An End Device cannot receive data packets directly, as it might be asleep when a packet arrives. The data packets for an End Device are therefore buffered by the device's parent and the End Device polls its parent for data while awake. An individual data packet is only held on the parent node for a maximum of 7.68 seconds and if many packets for the End Device are expected over a short period of time, the End Device should retrieve these packets as quickly as possible. An End Device can implement two polling modes, which are dependent on the poll interval (time-period between consecutive polls):

- **Normal poll mode:** A long poll interval is used - this mode is appropriate when the End Device is not expecting data packets.
- **Fast poll mode:** A short poll interval is used - this mode is appropriate when the End Device is expecting data packets.

The End Device may enable fast poll mode itself when it is expecting data packets (for example, after it has requested data from remote nodes). The Poll Control cluster allows fast poll mode to be selected from a remote control device to force the End Device to be more receptive to data packets (for example, when a download to the End Device involving a large number of unsolicited data packets is to be initiated).

The two sides of the cluster are located as follows:

- The cluster server is implemented on the End Device to be controlled
- The cluster client is implemented on the remote controller device

The cluster server (End Device) periodically checks whether the cluster client (remote controller) requires the poll mode to be changed. This 'check-in' method is used since an unsolicited instruction from the controller may arrive when the End Device is asleep. The automatic 'check-ins' are conducted with all the remote endpoints (on controller nodes) to which the local endpoint (on which the cluster resides) is bound.

The cluster is enabled by defining CLD_POLL_CONTROL in the **zcl_options.h** file. Further compile-time options for the Poll Control cluster are detailed in Section 20.10.

20.2 Cluster structure and attributes

The structure definition for the Poll Control cluster (server) is:

```
typedef struct
{
#ifdef POLL_CONTROL_SERVER
    zuint32     u32CheckinInterval;
    zuint32     u32LongPollInterval;
    zuint16     u16ShortPollInterval;
    zuint16     u16FastPollTimeout;
#ifdef CLD_POLL_CONTROL_ATTR_CHECKIN_INTERVAL_MIN
    zuint32     u32CheckinIntervalMin;
#endif
#ifdef CLD_POLL_CONTROL_ATTR_LONG_POLL_INTERVAL_MIN
    zuint32     u32LongPollIntervalMin;
#endif
#ifdef CLD_POLL_CONTROL_ATTR_FAST_POLL_TIMEOUT_MAX
    zuint16     u16FastPollTimeoutMax;
```

```
#endif
#endif
    zuint16     u16ClusterRevision;
} tsCLD_PollControl;
```

where:

- `u32CheckinInterval` is the ‘check-in interval’, used by the server in checking whether a client requires the poll mode to be changed - this is the period, in quarter-seconds, between consecutive checks. The valid range of values is 1 to 7208960. A user-defined minimum value for this attribute can be set via the optional attribute `u32CheckinIntervalMin` (see below). Zero is a special value indicating that the Poll Control cluster server is disabled. The default value is 14400 (1 hour).
- `u32LongPollInterval` is the ‘long poll interval’ of the End Device, employed when operating in normal poll mode - this is the period, in quarter-seconds, between consecutive polls of the parent for data. The valid range of values is 4 to 7208960. A user-defined minimum value for this attribute can be set via the optional attribute `u32LongPollIntervalMin` (see below). 0xFFFF is a special value indicating that the long poll interval is unknown/undefined. The default value is 20 (5 seconds).
- `u16ShortPollInterval` is the ‘short poll interval’ of the End Device, employed when operating in fast poll mode - this is the period, in quarter-seconds, between consecutive polls of the parent for data. The valid range of values is 1 to 65535 and the default value is 2 (0.5 seconds).
- `u16FastPollTimeout` is the ‘fast poll timeout’ representing the time-interval, in quarter-seconds, for which the server should normally stay in fast poll mode (unless over-ridden by a client command). The valid range of values is 1 to 65535. It is recommended that this timeout is greater than 7.68 seconds. A user-defined maximum value for this attribute can be set via the optional attribute `u16FastPollTimeoutMax` (see below). The default value is 40
(10 seconds).
- `u32CheckinIntervalMin` is an optional lower limit on the ‘check-in interval’ defined by `u32CheckinInterval`. This limit can be used to ensure that the interval is not inadvertently set to a low value which will quickly drain the energy resources of the End Device node.
- `u32LongPollIntervalMin` is an optional lower limit on the ‘long poll interval’ defined by `u32LongPollInterval`. This limit can be used to ensure that the interval is not inadvertently set (for example, by another device) to a low value which will quickly drain the energy resources of the End Device node.
- `u16FastPollTimeoutMax` is an optional upper limit on the ‘fast poll timeout’ defined by `u16FastPollTimeout`. This limit can be used to ensure that the interval is not inadvertently set (for example, by another device) to a high value which quickly drains the energy resources of the End Device node.

Note:

1. *Valid ranges (maximum and minimum values) for the four mandatory attributes can alternatively be set using macros in the **zcl_options.h** file, as described in Section 20.10. Some of these macros can only be used when the equivalent optional attribute is disabled.*
2. *For general guidance on attribute settings, refer to Section 20.3. Configuration through the attributes is also described in Section 20.4.2.*

u16ClusterRevision is a mandatory attribute that specifies the revision of the cluster specification on which this cluster instance is based. The cluster specification in the ZCL r6 corresponds to a cluster revision of 1. The value is incremented by one for each subsequent revision of the cluster specification. This attribute is also described in Section 2.4.

20.3 Attribute Settings

In assigning user-defined values to the mandatory attributes, the following inequality should be obeyed:

`u32CheckinInterval` ≥ `u32LongPollInterval` ≥ `u16ShortPollInterval`

In addition, the mandatory attribute `u16FastPollTimeout` should not be set to an excessive value for self-powered nodes, as fast poll mode can rapidly drain the stored energy of a node (for example, the battery).

The three optional attributes can be used to ensure that the values of the corresponding mandatory attributes are kept within reasonable limits, to prevent the rapid depletion of the energy resources of the node. If required, the optional attributes must be enabled and initialized in the compile-time options (see Section 20.10).

Minimum and maximum values for all the mandatory attributes can alternatively be set using the compile-time options (again, refer to Section 20.10).

20.4 Poll Control Operations

This section describes the main operations to be performed on the Poll Control cluster server (End Device) and client (controller).

20.4.1 Initialization

The Poll Control cluster must be initialized on both the cluster server and client. This can be done using the function **eCLD_PollControlCreatePollControl()**, which creates an instance of the Poll Control cluster on a local endpoint.

If you are using a standard ZigBee device which includes the Poll Control cluster, the above function is automatically called by the initialization function for the device. You only need to call **eCLD_PollControlCreatePollControl()** explicitly 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).

20.4.2 Configuration

When initialized, the Poll Control cluster adopts the attribute values that are pre-set in the `tsCLD_PollControl` structure (see Section 20.2). For the optional attributes, values can be set in the file **zcl_options.h** (see Section 20.10).

The mandatory attributes (and related optional attributes) are as follows:

- **Long Poll Interval (`u32LongPollInterval`)**: This is the polling period used in normal poll mode, expressed in quarter-seconds, with a default value of 20 (5 seconds). The attribute has a valid range of 4 to 7208960 but a user-defined minimum value for this attribute can be set via the optional ‘long poll interval maximum’ attribute (`u32LongPollIntervalMin`). This limit can be used to ensure that the interval is not inadvertently set (for example, by another device) to a low value which quickly drains the energy resources of the End Device node. Alternatively, minimum and maximum values can be specified through the compile-time options (see Section 20.10).
- **Short Poll Interval (`u16ShortPollInterval`)**: This is the polling period used in fast poll mode, expressed in quarter-seconds, with a default value of 2 (0.5 seconds). The attribute has a valid range of 1 to 65535. User-defined minimum and maximum values for this attribute can be specified through the compile-time options (see Section 20.10).
- **Fast Poll Timeout (`u16FastPollTimeout`)**: This is the time-interval for which the server should normally stay in fast poll mode (unless over-ridden by a client command), expressed in quarter-seconds, with a default value of 40 (10 seconds). It is recommended that this timeout is greater than 7.68 seconds. The valid range of values is 1 to 65535 but a user-defined maximum value for this attribute can be set via the optional ‘fast poll timeout maximum’ attribute (`u16FastPollTimeoutMax`). This limit can be used to ensure that the interval is not inadvertently set (for example, by another device) to a high value which quickly drains the energy resources of the End Device node. Alternatively, minimum and maximum values can be specified through the compile-time options (see Section 20.10).

- **Check-in Interval (`u32CheckinInterval`)**: This is the period between the server's checks of whether a client requires the poll mode to be changed, expressed in quarter-seconds, with a default value of 14400 (1 hour). It should be greater than the 'long poll interval' (see above). Zero is a special value indicating that the Poll Control cluster server is disabled. Otherwise, the valid range of values is 1 to 7208960 but a user-defined minimum value for this attribute can be set via the optional 'check-in interval minimum' attribute (`u32CheckinIntervalMin`). This limit can be used to ensure that the interval is not inadvertently set to a low value which quickly drains the energy resources of the End Device node. Alternatively, minimum and maximum values can be specified through the compile-time options (see Section 20.10).

The Poll Control cluster server can also be configured by the server application at run-time by writing to the relevant attribute(s) using the **eCLD_PollControlSetAttribute()** function (which must be called separately for each attribute to be modified). If used, this function must be called after the cluster has been initialized (see Section 20.4.1).

Changes to certain attributes can also be initiated remotely from the cluster client (controller) using the following functions:

- **eCLD_PollControlSetLongPollIntervalSend()**: The client application can use this function to submit a request to set the 'long poll interval' attribute on the server to a specified value. This function causes a 'Set Long Poll Interval' command to be sent to the relevant End Device. If the new value is acceptable, the cluster server automatically updates the attribute.
- **eCLD_PollControlSetShortPollIntervalSend()**: The client application can use this function to submit a request to set the 'short poll interval' attribute on the server to a specified value. This function causes a 'Set Short Poll Interval' command to be sent to the relevant End Device. If the new value is acceptable, the cluster server automatically updates the attribute.

In both of the above cases, a response is only sent back to the client if the new value is not acceptable, in which case a ZCL 'default response' is sent indicating an invalid value.

Use of the above two functions requires the corresponding commands to be enabled in the compile-time options, as described in Section 20.10.

Note: *Changes to attribute values initiated by either the server application or client application takes effect immediately. So, for example, if the End Device is operating in fast poll mode when the 'short poll interval' is modified, the polling period is immediately re-timed to the new value. If the modified attribute is not related to the currently operating poll mode, the change is implemented the next time the relevant poll mode is started.*

Before the first scheduled 'check-in' (after one hour, by default), the End Device application should set up bindings between the local endpoint on which the cluster resides and the relevant endpoint on each remote controller node with which the End Device operates. These bindings are used while sending the 'Check-in' commands.

20.4.3 Operation

After initialization, the Poll Control cluster server on the End Device begins to operate in normal poll mode and performs the following activities (while the End Device is awake):

- Periodically poll the parent for data packets at a rate determined by the 'long poll interval'.
- Periodically check whether any bound cluster clients require the server to enter fast poll mode, with 'check-ins' at a rate determined by the 'check-in interval'.

The server application must provide the cluster with timing prompts for the above periodic activities. These prompts are produced by periodically calling the function **eCLD_PollControlUpdate()**. The periods of the above activities are defined in terms of quarter-seconds. Therefore, this function must be called every quarter-second and the application must provide a 250 ms software timer to schedule these calls. Any poll or check-in that is due when this function is called is automatically performed by the cluster server.

The End Device operates in normal poll mode until either it puts itself into fast poll mode (for example, when it is expecting responses to a request) or the controller (client) requests the End Device to enter fast poll mode (for example, when a data download to the End Device is going to be performed). As indicated above, such a request from the client is raised as the result of the server performing periodic 'check-ins' with the client.

On receiving a 'check-in' command, an `E_CLD_POLL_CONTROL_CMD_CHECK_IN` event is generated on the client. The client application must then fill in the `tsCLD_PollControl_CheckinResponsePayload` structure (see Section 20.9.2) of the event, indicating whether fast poll mode is required. A response is then automatically sent back to the server.

After sending the initial Check-in command, the server waits for up to 7.68 seconds for a response (if no response is received in this time, the server is free to continue in normal poll mode). If a response is received from a client, the event `E_CLD_POLL_CONTROL_CMD_CHECK_IN` is generated on the server, where this event indicates the processing status of the received response. The server also sends this status back to the responding client in a ZCL default response.

- If the response was received from a bound client within the timeout period of the initial Check-in command, the status is ZCL_SUCCESS. In this case, the End Device is automatically put into fast poll mode.
- If the response is invalid for some reason, an error status is indicated as described below in Section 20.4.3.2, and fast poll mode is not entered.

When the End Device is in fast poll mode, the client application can request the cluster server to exit fast poll mode immediately (before the timeout expires) by calling the function **eCLD_PollControlFastPollStopSend()**.

20.4.3.1 Fast Poll Mode Timeout

In the Check-in response from a client, the payload (see Section 20.9.2) may contain an optional timeout value which, if used, specifies the length of time that the device should remain in fast poll mode (this timeout value will be used instead of the one specified through the 'fast poll timeout' attribute). If the response payload specifies an out-of-range timeout value, the server will send a ZCL default response with status INVALID_VALUE to the client (see Section 20.4.3.2). In the case of multiple clients (controllers) that have specified different timeout values, the server will use the largest timeout value received.

20.4.3.2 Invalid Check-in Responses

The server may receive Check-in responses which cannot result in fast poll mode. In these cases, the server sends a ZCL default response indicating the relevant error status (which is not ZCL_SUCCESS) back to the originating client. The following circumstances will lead to such a default response:

- The Check-in response is from an unbound client. In this case, the Default Response will contain the status ACTION_DENIED.
- The Check-in response is from a bound client but requests an invalid fast poll timeout value (see Section 20.4.3.1). In this case, the default response will contain the status INVALID_VALUE.
- The Check-in response is from a bound client but arrives after the timeout period of the original Check-in command. In this case, the default response will contain the status TIMEOUT.

20.5 Poll Control Events

The Poll Control cluster has its own events that are handled through the callback mechanism described in Chapter 3. The cluster contains its own event handler. However, if a device uses this cluster then application-specific Poll Control event handling must be included in the user-defined callback function for the associated endpoint, where this callback function is registered through the relevant endpoint registration function. This callback function will then be invoked when a Poll Control event occurs and needs the attention of the application.

For a Poll Control 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 `tsCLD_PollControlCallBackMessage` structure:

```
typedef struct
{
 uint8 u8CommandId;
 union
 {
 tsCLD_PollControl_CheckinResponsePayload *psCheckinResponsePayload;
 #ifdef CLD_POLL_CONTROL_CMD_SET_LONG_POLL_INTERVAL
 tsCLD_PollControl_SetLongPollIntervalPayload
            *psSetLongPollIntervalPayload;
 #endif
 #ifdef CLD_POLL_CONTROL_CMD_SET_SHORT_POLL_INTERVAL
       tsCLD_PollControl_SetShortPollIntervalPayload
    *psSetShortPollIntervalPayload;
 #endif
 } uMessage;
} tsCLD_PollControlCallBackMessage;
```

The above structure is fully described in Section 20.9.1.

When a Poll Control event occurs, one of the command types listed in Table 29 is specified through the `u8CommandId` field of the structure `tsCLD_PollControlCallBackMessage`. This command type determines which command payload is used from the union `uMessage`.

Table 39. Poll Control Command Types (Events)

u8CommandId Enumeration	Description/Payload Type
On Client	
E_CLD_POLL_CONTROL_CMD_CHECK_IN	A Check-in command has been received by the client.
On Server	
E_CLD_POLL_CONTROL_CMD_CHECK_IN	A Check-in Response has been received by the server, following a previously sent Check-In command. **tsCLD_PollControl_CheckinResponsePayload**
E_CLD_POLL_CONTROL_CMD_FAST_POLL_STOP	A 'Fast Poll Stop' command has been received by the server.
E_CLD_POLL_CONTROL_CMD_SET_LONG_POLL_INTERVAL	A 'Set Long Poll Interval' command has been received by the server. **tsCLD_PollControl_SetLongPollIntervalPayload**
E_CLD_POLL_CONTROL_CMD_SET_SHORT_POLL_INTERVAL	A 'Set Short Poll Interval' command has been received by the server. **tsCLD_PollControl_SetShortPollIntervalPayload**

20.6 Functions

The Poll Control cluster functions are described in the following three sub-sections, according to the side(s) of the cluster on which they can be used:

- Server/client functions are described in Section 20.6.1
- Server functions are described in Section 20.6.2
- Client functions are described in Section 20.6.3

20.6.1 Server/Client Function

The following Poll Control cluster function can be used on either a cluster server or cluster client:

eCLD_PollControlCreatePollControl

20.6.1.1 eCLD_PollControlCreatePollControl

```
teZCL_Status eCLD_PollControlCreatePollControl(
 tsZCL_ClusterInstance  *psClusterInstance,
  bool_t  bIsServer,
  tsZCL_ClusterDefinition  *psClusterDefinition,
  void *pvEndPointSharedStructPtr,
  uint8 *pu8AttributeControlBits,
 tsCLD_PollControlCustomDataStructure *psCustomDataStructure);
```

Description

This function creates an instance of the Poll Control 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 Poll Control 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 D.

Note: *This function must not be called for an endpoint on which a standard ZigBee device is 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 the ZigBee Devices User Guide (JNUG3131).*

When used, this function must be the first Poll Control cluster function called in the application, and must be called after the stack has been started and after the ZCL has been initialized.

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 is automatically adjusted by the compiler using the following declaration:

```
uint8 au8PollControlAttributeControlBits
[(sizeof(asCLD_PollControlClusterAttrDefs) / sizeof(tsZCL_AttributeDefinition))];
```

Parameters

- *psClusterInstance*: Pointer to structure containing information about the cluster instance to be created (see Section 6.1.16). This structure is updated by the function by initializing individual structure fields.
- *bIsServer*: Type of cluster instance (server or client) to be created:
- TRUE - server
- FALSE - client
- *psClusterDefinition*: Pointer to structure indicating the type of cluster to be created (see Section 6.1.2). In this case, this structure must contain the details of the Poll Control cluster. This parameter can refer to a pre-filled structure called `sCLD_PollControl` which is provided in the **PollControl.h** file.
- *pvEndPointSharedStructPtr*: Pointer to the shared structure used for attribute storage. This parameter should be the address of the structure of type `tsCLD_PollControl` which defines the attributes of the Poll Control cluster. The function initializes the attributes with default values.

- *pu8AttributeControlBits*: Pointer to an array of **uint8** values, with one element for each attribute in the cluster (see above). For a cluster client, set this pointer to NULL.
- *psCustomDataStructure*: Pointer to a structure containing the storage for internal functions of the cluster (see Section 20.9.5).

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_INVALID_VALUE

20.6.2 Server Functions

The following Poll Control cluster functions can be used on a cluster server only:

- eCLD_PollControlUpdate
- eCLD_PollControlSetAttribute
- eCLD_PollControlUpdateSleepInterval

20.6.2.1 eCLD_PollControlUpdate

```
teZCL_Status eCLD_PollControlUpdate(void);
```

Description

This function can be used on a cluster server to update the timing status for the following periodic activities:

- polling of the parent for a data packet
- ‘check-ins’ with the client to check for a required change in the poll mode

The function should be called once per quarter-second and the application should provide a 250-ms timer to prompt these function calls.

Any poll or check-in that is due when this function is called are automatically performed by the cluster server.

Parameters

None

Returns

E_ZCL_SUCCESS
E_ZCL_FAIL

20.6.2.2 eCLD_PollControlSetAttribute

```
teZCL_Status eCLD_PollControlSetAttribute(
    uint8 u8SourceEndPointId,
    uint8 u8AttributeId,
     uint32 u32AttributeValue);
```

Description

This function can be used on a cluster server to write to an attribute of the Poll Control cluster. The function writes to the relevant field of the `tsCLD_PollControl` structure (detailed in Section 20.2). The attribute to be accessed is specified using its attribute identifier - enumerations are provided (see Section 20.8.1).

Therefore, this function can be used to change the configuration of the Poll Control cluster. The change takes effect immediately. So, for example, if the End Device is in normal poll mode when the 'long poll interval' is modified, the polling period is immediately re-timed to the new value. If the modified attribute is not related to the currently operating poll mode, the change is implemented the next time the relevant poll mode is started.

The specified value of the attribute is validated by the function. If this value is out-of-range for the attribute, the status `E_ZCL_ERR_INVALID_VALUE` is returned.

Parameters

- *u8SourceEndPointId*: Number of local endpoint on which cluster resides
- *u8AttributeId* : Identifier of attribute to be written to (see Section 20.8.1)
- *u32AttributeValue*: Value to be written to attribute

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL
- E_ZCL_ERR_INVALID_VALUE
- E_ZCL_DENY_ATTRIBUTE_ACCESS

20.6.2.3 eCLD_PollControlUpdateSleepInterval

```
teZCL_Status eCLD_PollControlUpdateSleepInterval(
 uint32 u32QuarterSecondsAsleep);
```

Description

This function can be used on a cluster server to provide the updated ticks back into PollControl cluster for the time the device was sleeping in terms of quarter second.

This function updates the Checkin period based on the ticks provided.

Parameters

u32QuarterSecondsAsleep: Number of Quarter seconds the device has slept for

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_CLUSTER_NOT_FOUND

20.6.3 Client Functions

The following Poll Control cluster functions can be used on a cluster client only:

1. eCLD_PollControlSetLongPollIntervalSend
2. eCLD_PollControlSetShortPollIntervalSend
3. eCLD_PollControlFastPollStopSend

20.6.3.1 eCLD_PollControlSetLongPollIntervalSend

```
teZCL_Status eCLD_PollControlSetLongPollIntervalSend(
   uint8 u8SourceEndPointId,
   uint8 u8DestinationEndPointId,
   tsZCL_Address *psDestinationAddress,
   uint8 *pu8TransactionSequenceNumber,
   tsCLD_PollControl_SetLongPollIntervalPayload
 *psPayload);
```

Description

This function can be used on a cluster client to send a ‘Set Long Poll Interval’ command to the cluster server. This command requests the ‘long poll interval’ for normal poll mode on the End Device to be set to the specified value.

On receiving the command, the ‘long poll interval’ attribute is only modified by the server if the specified value is within the valid range for the attribute (including greater than or equal to the optional user-defined minimum, if set) - see Section 20.2. If this is not the case, the server replies to the client with a ZCL ‘default response’ indicating an invalid value.

The change takes effect immediately. So, if the End Device is in normal poll mode when the ‘long poll interval’ is modified, the polling period is immediately re-timed to the new value.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the message. The TSN in the response is 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 local endpoint on which cluster client resides
- *u8DestinationEndPointId*: Number of remote endpoint on which cluster server resides
- *psDestinationAddress*: Pointer to a structure containing the destination address of the server node
- *pu8TransactionSequenceNumber*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the message
- *psPayload*: Pointer to structure containing the payload for the command (see Section 20.9.3), including the desired long poll interval

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

20.6.3.2 eCLD_PollControlSetShortPollIntervalSend

```
teZCL_Status eCLD_PollControlSetShortPollIntervalSend(
  uint8 u8SourceEndPointId,
  uint8 u8DestinationEndPointId,
  tsZCL_Address *psDestinationAddress,
  uint8 *pu8TransactionSequenceNumber,
  tsCLD_PollControl_SetShortPollIntervalPayload *psPayload);
```

Description

This function can be used on a cluster client to send a ‘Set Short Poll Interval’ command to the cluster server. This command requests the ‘short poll interval’ for fast poll mode on the End Device to be set to the specified value.

On receiving the command, the ‘short poll interval’ attribute is only modified by the server if the specified value is within the valid range for the attribute (including greater than or equal to the optional user-defined minimum, if set) - see Section 20.2. If this is not the case, the server replies to the client with a ZCL ‘default response’ indicating an invalid value.

The change takes effect immediately. So, if the End Device is in fast poll mode when the ‘short poll interval’ is modified, the polling period is immediately re-timed to the new value.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the message. The TSN in the response is 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 local endpoint on which cluster client resides
- *u8DestinationEndPointId*: Number of remote endpoint on which cluster server resides
- *psDestinationAddress*: Pointer to a structure containing the destination address of the server node
- *pu8TransactionSequenceNumber*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the message
- *psPayload*: Pointer to a structure containing the payload for the command (see Section 20.9.4), including the desired short poll interval

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

20.6.3.3 eCLD_PollControlFastPollStopSend

```
teZCL_Status eCLD_PollControlFastPollStopSend(
 uint8 u8SourceEndPointId,
 uint8 u8DestinationEndPointId,
```

```
  tsZCL_Address *psDestinationAddress,
  uint8 *pu8TransactionSequenceNumber);
```

Description

This function can be used on a cluster client to send a ‘Fast Poll Stop’ command to the cluster server. This command is intended to abort a fast poll mode episode which has been started on the server as the result of a ‘Check-in Response’. Therefore, the command allows fast poll mode to be exited before the mode’s timeout is reached.

The cluster server only stops fast poll mode on the destination End Device if a matching ‘Fast Poll Stop’ command has been received for every request to start the current episode of fast poll mode. Therefore, if the current fast poll mode episode resulted from multiple start requests from multiple clients, the episode cannot be prematurely stopped (before the timeout is reached) unless a ‘Fast Poll Stop’ command is received from each of those clients.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the message. The TSN in the response is 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 local endpoints on which the cluster client resides
- *u8DestinationEndPointId*: Number of remote endpoints on which the cluster server resides
- *psDestinationAddress*: Pointer to a structure containing the destination address of the server node
- *pu8TransactionSequenceNumber*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the message

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

20.7 Return codes

The Poll Control cluster functions use the ZCL return codes, listed in Section 7.2.

20.8 Enumerations

20.8.1 ‘Attribute ID’ enumerations

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

```
 typedef enum PACK
{
 E_CLD_POLL_CONTROL_ATTR_ID_CHECKIN_INTERVAL 0x0000,
 E_CLD_POLL_CONTROL_ATTR_ID_LONG_POLL_INTERVAL,
```

```
 E_CLD_POLL_CONTROL_ATTR_ID_SHORT_POLL_INTERVAL,
 E_CLD_POLL_CONTROL_ATTR_ID_FAST_POLL_TIMEOUT,
 E_CLD_POLL_CONTROL_ATTR_ID_CHECKIN_INTERVAL_MIN,
 E_CLD_POLL_CONTROL_ATTR_ID_LONG_POLL_INTERVAL_MIN,
 E_CLD_POLL_CONTROL_ATTR_ID_FAST_POLL_TIMEOUT_MAX;
}teCLD_PollControl_Cluster_AttrID;
```

20.8.2 'Command' Enumerations

The following enumerations represent the commands that the Poll Control cluster generates.

```
typedef enum PACK
{
  E_CLD_POLL_CONTROL_CMD_CHECK_IN = 0x00,
  E_CLD_POLL_CONTROL_CMD_FAST_POLL_STOP,
  E_CLD_POLL_CONTROL_CMD_SET_LONG_POLL_INTERVAL,
  E_CLD_POLL_CONTROL_CMD_SET_SHORT_POLL_INTERVAL,
} teCLD_PollControl_CommandID;
```

The above enumerations are used to indicate types of Poll Control cluster events and are described in Section 20.5.

20.9 Structures

20.9.1 tsCLD_PPCallBackMessage

For a Poll Control 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 `tsCLD_PollControlCallBackMessage` structure:

```
typedef struct
{
 uint8 u8CommandId;
 union
 {
 tsCLD_PollControl_CheckinResponsePayload *psCheckinResponsePayload;
 #ifdef CLD_POLL_CONTROL_CMD_SET_LONG_POLL_INTERVAL
 tsCLD_PollControl_SetLongPollIntervalPayload
       *psSetLongPollIntervalPayload;
  #endif
  #ifdef CLD_POLL_CONTROL_CMD_SET_SHORT_POLL_INTERVAL
  tsCLD_PollControl_SetShortPollIntervalPayload
        *psSetShortPollIntervalPayload;
  #endif
 } uMessage;
} tsCLD_PollControlCallBackMessage;
```

where:

- `u8CommandId` indicates the type of Poll Control command that has been received, one of:
  - E_CLD_POLL_CONTROL_CMD_CHECK_IN
  - E_CLD_POLL_CONTROL_CMD_FAST_POLL_STOP
  - E_CLD_POLL_CONTROL_CMD_SET_LONG_POLL_INTERVAL
  - E_CLD_POLL_CONTROL_CMD_SET_SHORT_POLL_INTERVAL

If they are required, the last two commands must be enabled in the compile-time options, as described in Section 20.10.

- `uMessage` is a union containing the command payload, as one of (depending on the value of `u8CommandId`):
  - psCheckinResponsePayload is a pointer to the payload of a 'Check-in Response' (see Section 20.9.2)
  - psSetLongPollIntervalPayload is a pointer to the payload of a 'Set Long Poll Interval' command (see Section 20.9.3)
  - psSetShortPollIntervalPayload is a pointer to the payload of a 'Set Short Poll Interval' command (see Section 20.9.4)

The command payload for each command type is indicated in Table 29 in Section 20.5.

20.9.2 tsCLD_PollControl_CheckinResponsePayload

This structure contains the payload of a 'Check-in Response', which is sent from the client to the server in reply to a 'Check-in' command from the server.

```
typedef struct
{
  zbool   bStartFastPolling;
  zuint16  u16FastPollTimeout;
}tsCLD_PollControl_CheckinResponsePayload;
```

where:

- `bStartFastPolling` is a boolean indicating whether or not the End Device is required to enter fast poll mode:
  - TRUE: Enter fast poll mode
  - FALSE: Continue in normal poll mode
- `u16FastPollTimeout` is an optional fast poll mode timeout, in quarter-seconds, in the range 1 to 65535 - that is, the period of time for which the End Device should remain in fast poll mode (if this mode is requested through `bStartFastPolling`). Zero is a special value which indicates that the value of the 'fast poll timeout' attribute should be used instead (see Section 20.2). If a non-zero value is specified then this value over-rides the 'fast poll timeout' attribute (but does not over-write it).

20.9.3 tsCLD_PollControl_SetLongPollIntervalPayload

This structure contains the payload of a 'Set Long Poll Interval' command, which is sent from the client to the server to request a new 'long poll interval' for use in normal poll mode.

```
typedef struct
{
 zuint32  u32NewLongPollInterval;
}tsCLD_PollControl_SetLongPollIntervalPayload;
```

where `u32NewLongPollInterval` is the required value of the 'long poll interval', in quarter-seconds, in the range 4 to 7208960. This value is used to over-write the corresponding cluster attribute if the specified value is within the valid range for the attribute (including greater than or equal to the optional user-defined minimum, if set).

To use the 'Set Long Poll Interval' command, it must be enabled in the compile-time options, as described in Section 20.10.

20.9.4 tsCLD_PollControl_SetShortPollIntervalPayload

This structure contains the payload of a ‘Set Short Poll Interval’ command, which is sent from the client to the server to request a new ‘short poll interval’ for use in fast poll mode.

```
typedef struct
{
 zuint16  u16NewShortPollInterval;
}tsCLD_PollControl_SetShortPollIntervalPayload;
```

where `u16NewShortPollInterval` is the required value of the ‘short poll interval’, in quarter-seconds, in the range 1 to 65535. This value is used to over-write the corresponding cluster attribute if the specified value is within the valid range for the attribute (including greater than or equal to the optional user-defined minimum, if set).

To use the ‘Set Short Poll Interval’ command, it must be enabled in the compile-time options, as described in Section 20.10.

20.9.5 tsCLD_PollControlCustomDataStructure

The Poll Control cluster requires extra storage space to be allocated to be used by internal functions. The structure definition for this storage is shown below:

```
typedef struct
{
#ifdef POLL_CONTROL_SERVER
    tsCLD_PollControlParameters         sControlParameters;
#endif
    tsZCL_ReceiveEventAddress           sReceiveEventAddress;
    tsZCL_CallBackEvent                 sCustomCallBackEvent;
    tsCLD_PollControlCallBackMessage    sCallBackMessage;
} tsCLD_PollControlCustomDataStructure;
```

The fields are for internal use and no knowledge of them is required.

20.10 Compile-time Options

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

To enable the Poll Control cluster in the code to be built, it is necessary to add the following line to the file:

```
#define CLD_POLL_CONTROL
```

In addition, to enable the cluster as a client or server, it is also necessary to add one of the following lines to the same file:

```
#define POLL_CONTROL_SERVER
#define POLL_CONTROL_CLIENT
```

The following options can also be configured at compile-time in the **zcl_options.h** file.

Optional Server Attributes

To enable and assign a value (`t` quarter-seconds) to the optional Check-in Interval Minimum (`u32CheckinIntervalMin`) attribute, add the line:

```
#define CLD_POLL_CONTROL_ATTR_CHECKIN_INTERVAL_MIN t
```

To enable and assign a value (`t` quarter-seconds) to the optional Long Poll Interval Minimum (`u32LongPollIntervalMin`) attribute, add the line:

```
#define CLD_POLL_CONTROL_ATTR_LONG_POLL_INTERVAL_MIN t
```

To enable and assign a value (`t` quarter-seconds) to the optional Fast Poll Timeout Maximum (`u16FastPollTimeoutMax`) attribute, add the line:

```
#define CLD_POLL_CONTROL_ATTR_FAST_POLL_TIMEOUT_MAX t
```

Note: *For further information on the above optional server attributes, refer to Section 20.2.*

Global Attributes

Add this line to define the value (n) of the Cluster Revision attribute:

```
#define CLD_POLL_CONTROL_CLUSTER_REVISION <n>
```

The default value is 1, which corresponds to the revision of the cluster in the ZCL r6 specification (see Section 2.4).

Set Valid Range for ‘Check-in Interval’

To set the maximum possible ‘check-in interval’ (to `t` quarter-seconds), add the line:

```
#define CLD_POLL_CONTROL_CHECKIN_INTERVAL_MAX t
```

The default value is 7208960.

To set the minimum possible ‘check-in interval’ (to `t` quarter-seconds), add the line:

```
#define CLD_POLL_CONTROL_CHECKIN_INTERVAL_MIN t
```

The default value is 0.

This minimum value is only applied if the Check-in Interval Minimum attribute (`u32CheckinIntervalMin`) is not enabled.

Set Valid Range for ‘Fast Poll Timeout’

To set the maximum possible ‘fast poll timeout’ (to `t` quarter-seconds), add the line:

```
#define CLD_POLL_CONTROL_FAST_POLL_TIMEOUT_MAX t
```

The default value is 65535.

To set the minimum possible ‘fast poll timeout’ (to `t` quarter-seconds), add the line:

```
#define CLD_POLL_CONTROL_FAST_POLL_TIMEOUT_MIN t
```

The default value is 1.

This maximum value is only applied if the Fast Poll Timeout Maximum attribute (`u16FastPollTimeoutMax`) is not enabled.

Set Valid Range for ‘Long Poll Interval’

To set the maximum possible ‘long poll interval’ (to `t` quarter-seconds), add the line:

```
#define CLD_POLL_CONTROL_LONG_POLL_INTERVAL_MAX t
```

The default value is 7208960.

To set the minimum possible ‘long poll interval’ (to `t` quarter-seconds), add the line:

```
#define CLD_POLL_CONTROL_LONG_POLL_INTERVAL_MIN t
```

The default value is 4.

This minimum value is only applied if the Long Poll Interval Minimum attribute (`u32LongPollIntervalMin`) is not enabled.

Set Valid Range for ‘Short Poll Interval’

To set the maximum possible ‘short poll interval’ (to `t` quarter-seconds), add the line:

```
#define CLD_POLL_CONTROL_SHORT_POLL_INTERVAL_MAX t
```

The default value is 65535.

To set the minimum possible ‘short poll interval’ (to `t` quarter-seconds), add the line:

```
#define CLD_POLL_CONTROL_SHORT_POLL_INTERVAL_MIN t
```

The default value is 1.

Optional Commands

To enable the optional ‘Set Long Poll Interval’ command, add the line:

```
#define CLD_POLL_CONTROL_CMD_SET_LONG_POLL_INTERVAL
```

To enable the optional ‘Set Short Poll Interval’ command, add the line:

```
#define CLD_POLL_CONTROL_CMD_SET_SHORT_POLL_INTERVAL
```

Maximum Number of Clients

To set the maximum number of clients for a server to `n`, add the line:

```
#define CLD_POLL_CONTROL_NUMBER_OF_MULTIPLE_CLIENTS n
```

This is the maximum number of clients from which the server can handle Check-in Responses. It should be equal to the capacity (number of entries) of the binding table created on the server device to accommodate bindings to client devices (where this size is set in a ZigBee network parameter using the ZPS Configuration Editor).

Disable APS Acknowledgments for Bound Transmissions

To disable APS acknowledgments for bound transmissions from this cluster, add the line:

```
#define CLD_POLL_CONTROL_BOUND_TX_WITH_APS_ACK_DISABLED
```

21 Power Profile Cluster

This chapter describes the Power Profile cluster which provides an interface between a home appliance (e.g. a washing machine) and the controller of an energy management system.

The Power Profile cluster has a Cluster ID of 0x001A.

21.1 Overview

The Power Profile cluster allows an appliance, the cluster server, to provide its expected power usage data to a controller, the cluster client. This ‘power profile’ represents the predicted ‘energy footprint’ of the appliance, and may be used by the controller to schedule and control the operation of the appliance. It may be requested by the client or provided unsolicited by the server.

The cluster is enabled by defining CLD_PP in the **zcl_options.h** file. Further compile-time options for the Power Profile cluster are detailed in Section 21.11.

Note: *The Power Profile cluster requires the Appliance Control cluster for the implementation of status notifications and power management commands. The Appliance Control cluster is described in Chapter 45.*

21.2 Cluster structure and attributes

The structure definition for the Power Profile cluster (server) is:

```
typedef struct
{
#ifdef PP_SERVER
   zuint8  u8TotalProfileNum;
   zbool  bMultipleScheduling;
   zbmap8  u8EnergyFormatting;
   zbool  bEnergyRemote;
   zbmap8  u8ScheduleMode;
#ifdef CLD_PP_ATTR_ATTRIBUTE_REPORTING_STATUS
   zuint8 u8AttributeReportingStatus;
#endif
#endif
   zuint16  u16ClusterRevision;
} tsCLD_PP;
```

where:

- `u8TotalProfileNum` is the number of power profiles supported by the device (must be between 1 and 254, inclusive)
- `bMultipleScheduling` is a boolean indicating whether the server side of the cluster supports the scheduling of multiple energy phases or just a single energy phase at a time (according to commands received from the client):
  - TRUE if multiple energy phase scheduling is possible
  - FALSE if only single energy phase scheduling is possible
- `u8EnergyFormatting` indicates the format of the Energy fields in the Power Profile Notification and Power Profile Response:
  - Bits 0-2: Number of digits to the right of the decimal point
  - Bits 3-6: Number of digits to the left of the decimal point

  - Bit 7: If set to '1', any leading zeros are removed
- `bEnergyRemote` is a boolean indicating whether the cluster server (appliance) is configured for remote control (of energy management):
  - TRUE if at least one power profile is enabled for remote control
  - FALSE if no power profile is enabled for remote control

    This attribute is linked to the `bPowerProfileRemoteControl` field in the power profile record (see Section 21.10.13) - if the latter field is set to TRUE, the attribute is also automatically set to TRUE.
- `u8ScheduleMode` indicates the criterion (cheapest or greenest) that should be used by the cluster client (for example, energy management system) to schedule the power profiles:
  - 0x00 - criterion is left to the cluster server to choose
  - 0x01 - cheapest mode (minimize cost of energy usage)
  - 0x02 - greenest mode (maximize use of renewable energy sources)
  - 0x03 - compromise between cheapest and greenest

    All other values are reserved.
- `u8AttributeReportingStatus` is an optional attribute that should be enabled when attribute reporting is used for the cluster (see Section 2.3.5). The value of this attribute indicates whether there are attribute reports still pending (0x00) or the attribute reports are complete (0x01) - all other values are reserved. This attribute is also described in Section 2.4.
- `u16ClusterRevision` is a mandatory attribute that specifies the revision of the cluster specification on which this cluster instance is based. The cluster specification in the ZCL r6 corresponds to a cluster revision of 1. The value is incremented by one for each subsequent revision of the cluster specification. This attribute is also described in Section 2.4.

21.3 Attributes for default reporting

The following attributes of the Power Profile cluster can be selected for default reporting:

```
bEnergyRemote
u8ScheduleMode
```

Attribute reporting (including default reporting) is described in Appendix B. Enabling reports for these attributes is described in Appendix B.3.6.

21.4 Power profiles

An appliance can have one or more power profiles. An example of an appliance with multiple power profiles is a washing machine which has a number of programmes for different types of materials and loads.

Note: *The number of power profiles on a device must be defined in the file* ***zcl_options.h*** *(see Section 21.11).*

An individual power profile comprises a series of energy phases with different power demands. For example, these phases may correspond to the different cycles of a washing machine programme, such as wash, rinse, spin. Details of a power profile, including these energy phases, are held in an entry of the power profile table on the cluster server (appliance).

If the appliance is to be remotely controlled, the controller (cluster client) must 'learn' the details of the appliance's power profile so that it can control the scheduling of the energy phases. The schedule of a power profile is decided by the client, and includes energy phases and their relative start-times (the energy phases are not necessarily contiguous in time). A schedule is illustrated in Figure 5. The client must communicate the schedule for a power profile to the server where the schedule is executed.

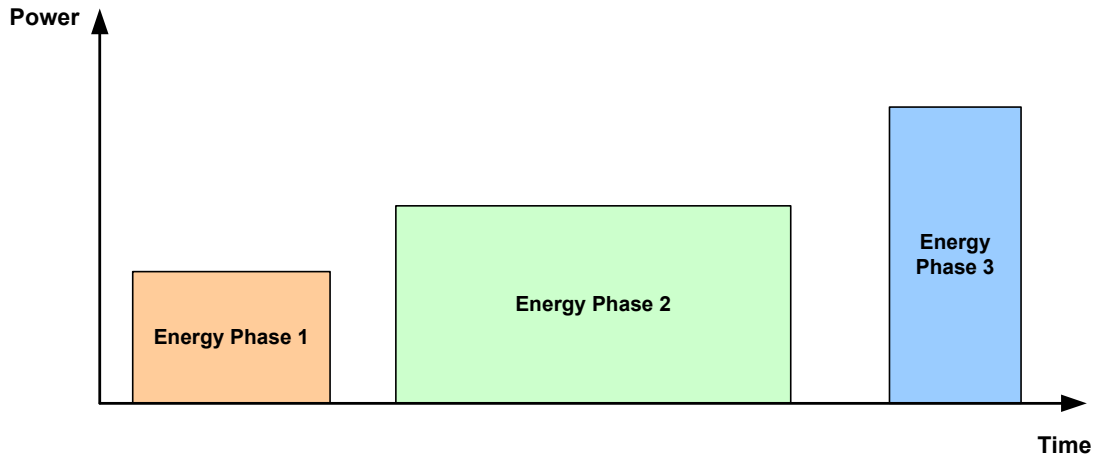


Figure 4. Schedule of Energy Phases of a Power Profile

21.5 Power profile operations

This section describes the main operations to be performed on the Power Profile cluster server (appliance) and client (controller).

21.5.1 Initialization

The Power Profile cluster must be initialized on both the cluster server and client. This can be done using the function **eCLD_PPCreatePowerProfile()**, which creates an instance of the Power Profile cluster on a local endpoint.

If you are using a Zigbee device that includes the Power Profile cluster, the above function is automatically called by the initialization function for the device. The function **eCLD_PPCreatePowerProfile()** should be called explicitly 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.

21.5.2 Adding and removing a power profile (server only)

A Power Profile cluster server (appliance) supports one or more power profiles. Information on these power profiles is held on the server in a power profile table, where each table entry contains information on one supported power profile.

The application on the appliance can perform various operations on the power profile table, as described in the sub-sections below.

21.5.2.1 Adding a power profile entry

The server application can introduce a new power profile by adding a corresponding entry to the power profile table using the function **eCLD_PPAddPowerProfileEntry()**. The new power profile table entry is specified in a `tsCLD_PPEntry` structure (see Section 21.10.2) supplied to this function. This structure includes the Power Profile ID - these identifiers should be numbered consecutively from 1 to 255.

The function **eCLD_PPAddPowerProfileEntry()** can also be used to replace (over-write) an existing power profile table entry, in which case the new entry should have the same Power Profile ID as the existing entry to be replaced.

21.5.2.2 Removing a power profile entry

The server application can remove a power profile from the device by calling the function **eCLD_PPRemovePowerProfileEntry()** to delete the corresponding entry of the local power profile table. The entry to be deleted is specified by means of the relevant Power Profile ID.

21.5.2.3 Obtaining a Power Profile Entry

The server application can obtain the details of a power profile supported by the server by reading the corresponding entry of the power profile table using the function **eCLD_PPGetPowerProfileEntry()**. The required entry is specified by means of the relevant Power Profile ID.

21.5.3 Communicating power profiles

In order to control the power consumption of the appliance (by scheduling the energy phases of the power profile), the controller (cluster client) must 'learn' the power profiles supported by the appliance (server). This may be done through requests or notifications, as described in the sub-sections below.

Note: *In order remotely control the appliance from a controller for energy management, the attribute* `bEnergyRemote` *of the Power Profile cluster on the server device must be set to TRUE (see Section 21.2).*

21.5.3.1 Requesting a power profile (by client)

A client application can request a power profile supported by the server by calling the **eCLD_PPPowerProfileReqSend()** function, which sends a Power Profile Request to the server. The function can be used to request a specific power profile (specified using its Power Profile ID) or all the power profiles supported by the server.

On receiving a response from the server, an E_CLD_PP_CMD_POWER_PROFILE_RSP event is generated on the client for each energy phase within the power profile. The reported information is contained in a `tsCLD_PP_PowerProfilePayload` structure (see Section 21.10.4). The application may store or discard this information, as required. By receiving the energy phase information in individual events, the application only needs to use as much memory as is required to store the relevant energy phase data.

Note: *The client application may first use the function* ***eCLD_PPPowerProfileStateReqSend()*** *to request the identifiers of the power profiles that are currently supported on the server.*

21.5.3.2 Notification of a power profile (by server)

The cluster server may send unsolicited notifications of the power profiles that it supports to the client. To do this, the server application must call the function **eCLD_PPPowerProfileNotificationSend()** which sends a Power Profile Notification containing the essential details of one supported power profile (such as the energy phases within the profile). This information is supplied to the function in a `tsCLD_PP_PowerProfilePayload` structure (see Section 21.10.4). If the server supports multiple power profiles, a separate notification must be sent for each profile.

On receiving the notification on the client, the event E_CLD_PP_CMD_POWER_PROFILE_NOTIFICATION is generated on the client for each energy phase within the power profile. The reported information is contained in a `tsCLD_PP_PowerProfilePayload` structure (see Section 21.10.4). The application may store or discard this information, as required. By receiving the energy phase information in individual events, the application only needs to use as much memory as is required to store the relevant energy phase data.

21.5.4 Communicating schedule information

A power profile schedule comprises a sequence of energy phases and their relative start-times (the energy phases may have gaps between them):

- An energy phase is identified by its Energy Phase Identifier, in the range 1 to 255 (inclusive).
- The start-time of an energy phase is expressed as a delay, in minutes, from the end of the previous energy phase. For the first energy phase of a power profile schedule, this delay is measured from the time that the schedule was started.

Note: *The normal duration of an energy phase, in minutes, is fixed and is specified in the energy phase information in the power profile.*

Although a power profile on the cluster server may support multiple energy phases, the schedule for the power profile may possibly incorporate only a sub-set of these phases. The client (controller) selects the set of energy phases in a schedule and communicates this schedule to the server (appliance). This may be done through a request or notification, as described in Section 21.5.4.1 and Section 21.5.4.2 below.

21.5.4.1 Requesting a schedule (by server)

The server application can request a schedule for a supported power profile from the client by calling the function **eCLD_PPEnergyPhasesScheduleReqSend()**, which sends an Energy Phases Schedule Request to the client.

The client can only return the requested schedule information if it stores this type of information for the power profile. If this is the case, an E_CLD_PP_CMD_ENERGY_PHASES_SCHEDULE_REQ event is generated on the client, with the `bIsInfoAvailable` field set to TRUE in the event structure `tsCLD_PPCallBackMessage`, and the client will send an Energy Phases Schedule Response back to the server. Otherwise, the client sends a ZCL default response with status NOT_FOUND.

On receiving an Energy Phases Schedule Response from the client, the event E_CLD_PP_CMD_ENERGY_PHASES_SCHEDULE_RSP is generated on the server, containing the requested schedule information in a `tsCLD_PP_EnergyPhasesSchedulePayload` structure (see Section 21.10.6). One or more of the following outcomes result:

- If the attribute `bEnergyRemote` is set to FALSE on the server (no remote control of the device), the server simply rejects the received schedule.
- If the received schedule information contains an `u16MaxActivationDelay` value of zero for an energy phase (see Section 21.10.11), this energy phase is rejected by the server although other valid energy phases are accepted. For each rejected energy phase, the server sends a ZCL default response with status NOT_AUTHORIZED to the client.
- If the received schedule information results in an update of the power profile schedule on the server, the server automatically sends an Energy Phases Schedule State Notification back to the client. On receiving this notification, an E_CLD_PP_CMD_ENERGY_PHASES_SCHEDULE_STATE_NOTIFICATION event is generated on the client.

Note: *Before requesting a power profile schedule, the server application may send the schedule's timing constraints to the client using the function* ***eCLD_PPPowerProfileScheduleConstraintsNotificationSend()****. The client application can alternatively request these schedule constraints from the server by calling* ***eCLD_PPPowerProfileScheduleConstraintsReqSend()****.*

21.5.4.2 Notification of a Schedule (by Client)

The cluster client may send an unsolicited notification of a power profile schedule to the server. To do this, the client application must call the function **eCLD_PPEnergyPhasesScheduleNotificationSend()** which sends an

Energy Phases Schedule Notification containing the schedule. This information is supplied to the function in a `tsCLD_PP_EnergyPhasesSchedulePayload` structure (see Section 21.10.6).

On receiving the notification on the server, the event E_CLD_PP_CMD_ENERGY_PHASES_SCHEDULE_NOTIFICATION is generated, containing the sent power profile schedule. One or more of the following outcomes will result:

- If the attribute `bEnergyRemote` is set to FALSE on the server (no remote control of the device), the server will simply reject the received schedule.
- If the received schedule information contains an `u16MaxActivationDelay` value of zero for an energy phase (see Section 21.10.11), this energy phase is rejected by the server although other valid energy phases will be accepted. For each rejected energy phase, the server will send a ZCL default response with status NOT_AUTHORIZED to the client.
- If the received schedule information results in an update of the power profile schedule on the server, the server will automatically send an Energy Phases Schedule State Notification back to the client. On receiving this notification, an E_CLD_PP_CMD_ENERGY_PHASES_SCHEDULE_STATE_NOTIFICATION event is generated on the client.

21.5.4.3 Notification of Energy Phases in Power Profile Schedule (by Server)

The server application can use the function **eCLD_PPEnergyPhasesScheduleStateNotificationSend()** to send an unsolicited Energy Phases Schedule State Notification to a cluster client, in order to inform the client of the energy phases that are in the schedule of a particular power profile.

21.5.4.4 Requesting the Scheduled Energy Phases (by Client)

The client application can use the function **eCLD_PPEnergyPhasesScheduleStateReqSend()** to send an Energy Phases Schedule State Request to the cluster server, in order to obtain the schedule of energy phases for a particular power profile on the server.

On receiving the response on the client, the event E_CLD_PP_CMD_ENERGY_PHASES_SCHEDULE_STATE_RSP is generated, containing the requested schedule information. The obtained schedule can be used to re-align the schedule information on the client with the information on the server - for example, after a reset of the client device.

21.5.5 Executing a Power Profile Schedule

After receiving a power profile schedule from the client (as described in Section 21.5.4), the server can start execution of the schedule. The instruction to start the schedule comes from the client in the form of an Energy Phases Schedule Notification. To issue this instruction, the client application must call the function **eCLD_PPEnergyPhasesScheduleNotificationSend()**. On receiving the notification, the server automatically starts the schedule.

The possible states of a power profile are fully detailed in Section 21.9.2 but, generally, it moves through the following principal states before, during and after execution:

- E_CLD_PP_STATE_PROGRAMMED: The power profile is defined in the local power profile table but a schedule has not been received from the client. Even without a schedule from the client, a schedule of energy phases that was defined when the power profile was introduced using the function **eCLD_PPAddPowerProfileEntry()** can be started from this state (see below).
- E_CLD_PP_STATE_WAITING_TO_START: The power profile remains in this state before the first energy phase starts and between energy phases (provided there is a gap between the end of one phase and the beginning of the next).
- E_CLD_PP_STATE_RUNNING: An energy phase is running.
- E_CLD_PP_STATE_ENDED: The final energy phase has completed.

Once a schedule has started, the server application must progress execution through the different states of the schedule by periodically calling the function **eCLD_PPSchedule()** once per second. This function moves the power profile to the next state, if it is due to start, and update the relevant state and timing parameters.

Note: *The server application can also use the function **eCLD_PPSetPowerProfileState()** to 'manually' move execution of the schedule to a particular (valid) state, irrespective of whether the target state is scheduled. This function can be used by the server application to locally start a schedule from the 'programmed' state.*

Whenever there is a change of state of a power profile, the cluster server automatically sends a Power Profile State Notification to the client. The server application can also send such a notification 'manually' by calling the function **eCLD_PPPowerProfileStateNotificationSend()**. The notification contains a power profile record that specifies the active power profile, the energy phase that is currently running (or due to run next) and the current state of the power profile. These notifications allow the controller to monitor the appliance. On receiving a notification on the client, an E_CLD_PP_CMD_POWER_PROFILE_STATE_NOTIFICATION event is generated, containing the sent power profile state information in a `tsCLD_PP_PowerProfileStatePayload` structure (see Section 21.10.5).

21.5.6 Communicating Price Information

The cost of implementing a power profile schedule on an appliance (cluster server) is determined/calculated by the controller (cluster client). The server can request price information from the client in a number of ways, as described below.

Note: *Use of the Power Profile Price functions, referenced below, must be enabled in the compile-time options, as described in Section 21.11.*

21.5.6.1 Requesting Cost of a Power Profile Schedule (by Server)

The server application can use the function **eCLD_PPGetPowerProfilePriceSend()** to send a Get Power Profile Price Request to the client, in order to request the cost of executing the schedule of a particular power profile.

The client can only return the requested information if price-related information about the power profile is held on the client device. If this is the case, an E_CLD_PP_CMD_GET_POWER_PROFILE_PRICE event is generated on the client, with the `bIsInfoAvailable` field set to TRUE in the event structure `tsCLD_PPCallBackMessage` and the client sends a Get Power Profile Price Response back to the server. Otherwise, the client sends a ZCL default response with status NOT_FOUND.

On receiving a Get Power Profile Price Response on the server, the event E_CLD_PP_CMD_GET_POWER_PROFILE_PRICE_RSP is generated, containing the requested price information (if available).

Alternatively, the server application can use the function **eCLD_PPGetPowerProfilePriceExtendedSend()** to send a Get Power Profile Price Extended Request to a cluster client, in order to request specific cost information about a power profile supported by the server. The cost of executing a power profile can be requested with either scheduled energy phases or contiguous energy phases (no gaps between them). This request is handled by the client as described above for an ordinary Get Power Profile Price Request. However, the response results in an E_CLD_PP_CMD_GET_POWER_PROFILE_PRICE_EXTENDED_RSP event on the server, containing the requested price information (if available).

21.5.6.2 Requesting Cost of Power Profile Schedules Over a Day (by Server)

The server application can use the **eCLD_PPGetOverallSchedulePriceSend()** function to send a Get Overall Schedule Price Request to the client, in order to obtain the overall cost of all the power profiles that will be executed over the next 24 hours.

The client can only return the requested information if price-related information about the relevant power profiles is held on the client device. If this is the case, an E_CLD_PP_CMD_GET_OVERALL_SCHEDULE_

PRICE event is generated on the client, with the `bIsInfoAvailable` field set to TRUE in the event structure `tsCLD_PPCallBackMessage`. Otherwise, the client will generate a ZCL default response with status NOT_FOUND.

On receiving a Get Overall Schedule Price Response on the server, the event E_CLD_PP_CMD_GET_OVERALL_SCHEDULE_PRICE_RSP is generated, containing the requested price information (if available).

21.6 Power Profile Events

The Power Profile cluster has its own events that are handled through the callback mechanism described in Chapter 3. The cluster contains its own event handler. However, if a device uses this cluster then application-specific Power Profile event handling must be included in the user-defined callback function for the associated endpoint, where this callback function is registered through the relevant endpoint registration function. This callback function is then invoked when a Power Profile event occurs and needs the attention of the application.

For a Power Profile 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 `tsCLD_PPCallBackMessage` structure:

```
typedef struct
{
 uint8  u8CommandId;
#ifdef PP_CLIENT
  bool  bIsInfoAvailable;
#endif
  union
   {
   tsCLD_PP_PowerProfileReqPayload
                  *psPowerProfileReqPayload;
   tsCLD_PP_GetPowerProfilePriceExtendedPayload
       *psGetPowerProfilePriceExtendedPayload;
   } uReqMessage;
   union
   {
   tsCLD_PP_GetPowerProfilePriceRspPayload
                  *psGetPowerProfilePriceRspPayload;
   tsCLD_PP_GetOverallSchedulePriceRspPayload
       *psGetOverallSchedulePriceRspPayload;
    tsCLD_PP_EnergyPhasesSchedulePayload
                  *psEnergyPhasesSchedulePayload;
    tsCLD_PP_PowerProfileScheduleConstraintsPayload
       *psPowerProfileScheduleConstraintsPayload;
    tsCLD_PP_PowerProfilePayload
                  *psPowerProfilePayload;
        tsCLD_PP_PowerProfileStatePayload
                  *psPowerProfileStatePayload;
    }uRespMessage;
} tsCLD_PPCallBackMessage;
```

The above structure is fully described in Section 21.10.1.

When a Power Profile event occurs, one of the command types listed in Table 30 and Table 31 is specified through the `u8CommandId` field of the `tsCLD_PPCallBackMessage` structure. This command type determines which command payload is used from the unions `uReqMessage` (for request commands) and `uRespMessage` (for response and notification commands).

Table 40. Power Profile Command Types (Events on Server)

u8CommandId Enumeration	Description/Payload Type
E_CLD_PP_CMD_POWER_PROFILE_REQ	The server (appliance) receives a Power Profile Request. **tsCLD_PP_PowerProfileReqPayload**

Table 40. Power Profile Command Types (Events on Server)*...continued*

u8CommandId Enumeration	Description/Payload Type
E_CLD_PP_CMD_POWER_PROFILE_STATE_REQ	The server (appliance) receives a Power Profile State Request.
E_CLD_PP_CMD_GET_POWER_PROFILE_PRICE_RSP	The server (appliance) receives a Get Power Profile Price Response, following a previously sent Get Power Profile Price Request. **tsCLD_PP_GetPowerProfilePriceRspPayload**
E_CLD_PP_CMD_GET_POWER_PROFILE_PRICE_EXTENDED_RSP	The server (appliance) receives a Get Power Profile Price Extended Response, following a previously sent Get Power Profile Price Extended Request. **tsCLD_PP_GetPowerProfilePriceRspPayload**
E_CLD_PP_CMD_GET_OVERALL_SCHEDULE_PRICE_RSP	The server (appliance) receives a Get Overall Schedule Price Response, following a previously sent Get Overall Schedule Price Request. **tsCLD_PP_GetOverallSchedulePriceRspPayload**
E_CLD_PP_CMD_ENERGY_PHASES_SCHEDULE_NOTIFICATION	The server (appliance) receives an Energy Phases Schedule Notification. **tsCLD_PP_EnergyPhasesSchedulePayload**
E_CLD_PP_CMD_ENERGY_PHASES_SCHEDULE_RSP	The server (appliance) receives an Energy Phases Schedule Response, following a previously sent Energy Phases Schedule Request. **tsCLD_PP_EnergyPhasesSchedulePayload**
E_CLD_PP_CMD_ENERGY_PHASES_SCHEDULE_STATE_RSP	The server (appliance) receives an Energy Phases Schedule State Response, following a previously sent Energy Phases Schedule State Request. **tsCLD_PP_EnergyPhasesSchedulePayload**
E_CLD_PP_CMD_GET_POWER_PROFILE_SCHEDULE_CONSTRAINTS_REQ	The server (appliance) receives a Get Power Profile Schedule Constraints Request. **tsCLD_PP_PowerProfileReqPayload**
E_CLD_PP_CMD_ENERGY_PHASES_SCHEDULE_STATE_REQ	The server (appliance) receives an Energy Phases Schedule State Request. **tsCLD_PP_PowerProfileReqPayload**

Table 41. Power Profile Command Types (Events on Client)

u8CommandId Enumeration	Description/Payload Type
E_CLD_PP_CMD_POWER_PROFILE_NOTIFICATION	The client (controller) receives a Power Profile Notification. **tsCLD_PP_PowerProfilePayload**
E_CLD_PP_CMD_POWER_PROFILE_STATE_NOTIFICATION	The client (controller) receives a Power Profile State Notification. **tsCLD_PP_PowerProfileStatePayload**
E_CLD_PP_CMD_POWER_PROFILE_RSP	The client (controller) receives a Power Profile Response, following a previously sent Power Pro-file Request. **tsCLD_PP_PowerProfilePayload**
E_CLD_PP_CMD_POWER_PROFILE_STATE_RSP	The client (controller) receives a Power Profile State Response, following a previously sent Power Profile State Request. **tsCLD_PP_PowerProfileStatePayload**
E_CLD_PP_CMD_GET_POWER_PROFILE_PRICE	The client (controller) receives a Get Power Profile Price Request. **tsCLD_PP_PowerProfileReqPayload**
E_CLD_PP_CMD_GET_OVERALL_SCHEDULE_PRICE	The client (controller) receives a Get Overall Schedule Price Request.

Table 41. Power Profile Command Types (Events on Client)*...continued*

u8CommandId Enumeration	Description/Payload Type
E_CLD_PP_CMD_ENERGY_PHASES_SCHEDULE_REQ	The client (controller) receives an Energy Phases Schedule Request . **tsCLD_PP_PowerProfileReqPayload**
E_CLD_PP_CMD_ENERGY_PHASES_SCHEDULE_STATE_NOTIFICATION	The client (controller) receives an Energy Phases Schedule State Notification. **tsCLD_PP_EnergyPhasesSchedulePayload**
E_CLD_PP_CMD_SCHEDULE_CONSTRAINTS_NOTIFICATION	The client (controller) receives a Power Profile Schedule Constraints Notification. **tsCLD_PP_PowerProfileScheduleConstraintsPayload**
E_CLD_PP_CMD_GET_POWER_PROFILE_SCHEDULE_CONSTRAINTS_RSP	The client (controller) receives a Power Profile Schedule Constraints Response. **tsCLD_PP_PowerProfileScheduleConstraintsPayload**
E_CLD_PP_CMD_GET_POWER_PROFILE_PRICE_EXTENDED	The client (controller) receives a Get Power Profile Price Extended Request. **tsCLD_PP_GetPowerProfilePriceExtendedPayload**

21.7 Functions

The Power Profile cluster functions are described in the following three sub-sections, according to the side(s) of the cluster on which they can be used:

- Server/client functions are described in Section 21.7.1
- Server functions are described in Section 21.7.2
- Client functions are described in Section 21.7.3

21.7.1 Server/Client Function

The following Power Profile cluster function can be used on either a cluster server or cluster client:

eCLD_PPCreatePowerProfile

21.7.1.1 eCLD_PPCreatePowerProfile

```
teZCL_Status eCLD_PPCreatePowerProfile(
  tsZCL_ClusterInstance *psClusterInstance,
  bool_t  bIsServer,
  tsZCL_ClusterDefinition   *psClusterDefinition,
  void *pvEndPointSharedStructPtr,
  uint8 *pu8AttributeControlBits,
  tsCLD_PPCustomDataStructure *psCustomDataStructure);
```

Description

This function creates an instance of the Power Profile 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 creates a Power Profile 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 D.

Note: *This function must not be called for an endpoint on which a standard Zigbee device is 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 the Zigbee Devices User Guide (JNUG3131).*

When used, this function must be the first Power Profile cluster function called in the application, and must be called after the stack has been started and after the ZCL has been initialized.

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 is automatically adjusted by the compiler using the following declaration:

```
uint8 au8PPAttributeControlBits[(sizeof(asCLD_PPClusterAttrDefs)/
        sizeof(tsZCL_AttributeDefinition))];
int8 au8PPAttributeControlBits[(sizeof(asCLD_PPClusterAttrDefs)/
        sizeof(tsZCL_AttributeDefinition))];
```

Parameters

psClusterInstance: Pointer to structure containing information about the cluster instance to be created (see Section 6.1.16). This structure is updated by the function by initializing individual structure fields.
bIsServer: Type of cluster instance (server or client) to be created:
TRUE - server
FALSE - client
psClusterDefinition: Pointer to structure indicating the type of cluster to be created (see Section 6.1.2). In this case, this structure must contain the details of the Power Profile cluster. This parameter can refer to a pre-filled structure called `sCLD_PP` which is provided in the **PowerProfile.h** file.
pvEndPointSharedStructPtr: Pointer to the shared structure used for attribute storage. This parameter should be the address of the structure of type `tsCLD_PP`, which defines the attributes of the Power Profile cluster. The function initializes the attributes with default values.
pu8AttributeControlBits: Pointer to an array of **uint8** values, with one element for each attribute in the cluster (see above). For a cluster client, set this pointer to NULL.
psCustomDataStructure: Pointer to a structure containing the storage for internal functions of the cluster (see Section 21.10.14)

Returns

E_ZCL_SUCCESS
E_ZCL_FAIL
E_ZCL_ERR_PARAMETER_NULL
E_ZCL_ERR_INVALID_VALUE

21.7.2 Server Functions

The following Power Profile cluster functions can be used on a cluster server only:

- eCLD_PPSchedule
- eCLD_PPSetPowerProfileState
- eCLD_PPAddPowerProfileEntry
- eCLD_PPRemovePowerProfileEntry
- eCLD_PPGetPowerProfileEntry
- eCLD_PPPowerProfileNotificationSend
- eCLD_PPEnergyPhaseScheduleStateNotificationSend

- eCLD_PPPowerProfileScheduleConstraintsNotificationSend
- eCLD_PPEnergyPhasesScheduleReqSend
- eCLD_PPPowerProfileStateNotificationSend
- eCLD_PPGetPowerProfilePriceSend
- eCLD_PPGetPowerProfilePriceExtendedSend
- eCLD_PPGetOverallSchedulePriceSend

21.7.2.1 eCLD_PPSchedule

```
teZCL_Status eCLD_PPSchedule(void);
```

Description

This function can be used on a cluster server to update the state of the currently active power profile and the timings required for scheduling. When called, the function automatically makes any required changes according to the scheduled energy phases for the power profile. If no change is scheduled, the function only updates timing information. If a change is required, it also updates the power profile state and the ID of the energy phase currently being executed.

The function should be called once per second to progress the active power profile schedule and the application should provide a 1-second timer to prompt these function calls.

Parameters

None

Returns

E_ZCL_SUCCESS
E_ZCL_FAIL
E_ZCL_ERR_PARAMETER_NULL
E_ZCL_ERR_INVALID_VALUE

21.7.2.2 eCLD_PPSetPowerProfileState

```
teZCL_CommandStatus eCLD_PPSetPowerProfileState(
   uint8 u8SourceEndPointId,
   uint8 u8PowerProfileId,
   teCLD_PP_PowerProfileState sPowerProfileState);
```

Description

This function can be used on a cluster server to move the specified power profile to the specified target state. Enumerations for the possible states are provided, and are listed and described in Section 21.9.2.

The function performs the following checks:

- Checks whether the specified Power Profile ID exists (if not, the function returns with the status E_ZCL_CMDS_NOT_FOUND)
- Checks whether the specified target state is a valid state (if not, the function returns with the status E_ZCL_CMDS_INVALID_VALUE)

- Checks whether the power profile is currently able move to the target state (if not, the function returns with the status E_ZCL_CMDS_INVALID_FIELD)

Note: *The power profile state can be changed by this function only if the move from the existing state to the target state is a valid change.*

If all the checks are successful, the move is implemented (and the function returns with the status E_ZCL_CMD_SUCCESS).

Parameters

u8SourceEndPointId: Number of local endpoint on which cluster resides
u8PowerProfileId : Identifier of the power profile
sPowerProfileState: Target state to which power profile is to be moved - enumerations are provided (see Section 21.9.2)

Returns

E_ZCL_CMD_SUCCESS
E_ZCL_CMDS_NOT_FOUND
E_ZCL_CMDS_INVALID_VALUE
E_ZCL_CMDS_INVALID_FIELD

21.7.2.3 eCLD_PPAddPowerProfileEntry

```
teZCL_Status eCLD_PPAddPowerProfileEntry(
  uint8 u8SourceEndPointId,
  tsCLD_PPEntry  *psPowerProfileEntry);
```

Description

This function can be used on a cluster server to introduce a new power profile by adding an entry to the local power profile table.

The function checks whether there is sufficient space in the table for the new power profile entry (if not, the function returns with the status E_ZCL_ERR_INSUFFICIENT_SPACE).

An existing power profile entry can be over-written with a new profile by specifying the same Power Profile ID (in the new entry structure).

The function also updates two of the cluster attributes (if needed), as follows.

- If a power profile is introduced which has multiple energy phases (as indicated by the `u8NumOfScheduledEnergyPhases` field of the `tsCLD_PPEntry` structure), the attribute `bMultipleScheduling` is set to TRUE (if not already TRUE)
- If a power profile is introduced which allows remote control for energy management (as indicated by the `bPowerProfileRemoteControl` field of the `tsCLD_PPEntry` structure), the attribute `bEnergyRemote` is set to TRUE (if not already TRUE)

Parameters

u8SourceEndPointId: Number of local endpoint on which cluster resides:
psPowerProfileEntry : Structure containing the power profile to add (see Section 21.10.2)

Returns

E_ZCL_SUCCESS
E_ZCL_FAIL
E_ZCL_ERR_PARAMETER_NULL
E_ZCL_ERR_INSUFFICIENT_SPACE

21.7.2.4 eCLD_PPRemovePowerProfileEntry

```
teZCL_Status eCLD_PPRemovePowerProfileEntry(
  uint8 u8SourceEndPointId,
  uint8 u8PowerProfileId);
```

Description

This function can be used on a cluster server to remove a power profile by deleting the relevant entry in the local power profile table.

Parameters

u8SourceEndPointId: Number of local endpoint on which cluster resides:
u8PowerProfileId : Identifier of power profile to be removed

Returns

E_ZCL_SUCCESS
E_ZCL_FAIL

21.7.2.5 eCLD_PPGetPowerProfileEntry

```
teZCL_Status eCLD_PPGetPowerProfileEntry(
        uint8 u8SourceEndPointId,
        uint8 u8PowerProfileId,
        tsCLD_PPEntry **ppsPowerProfileEntry);
```

Description

This function can be used on a cluster server to obtain an entry from the local power profile table. The required entry is specified using the relevant Power Profile ID. The function obtains a pointer to the relevant entry, if it exists - a pointer must be provided to a location to receive the pointer to the entry.

If no entry with the specified Power Profile ID is found, the function returns E_ZCL_ERR_INVALID_VALUE.

Parameters

u8SourceEndPointId: Number of local endpoint on which cluster resides
u8PowerProfileId: Identifier of power profile to be obtained
ppsPowerProfileEntry: Pointer to a location to receive a pointer to the required power profile table entry (see Section 21.10.2)

Returns

E_ZCL_SUCCESS
E_ZCL_FAIL
E_ZCL_ERR_INVALID_VALUE

21.7.2.6 eCLD_PPPowerProfileNotificationSend

```
teZCL_Status eCLD_PPPowerProfileNotificationSend(
     uint8 u8SourceEndPointId,
     uint8 u8DestinationEndPointId,
     tsZCL_Address *psDestinationAddress,
     uint8 *pu8TransactionSequenceNumber,
     tsCLD_PP_PowerProfilePayload *psPayload);
```

Description

This function can be used on the cluster server to send a Power Profile Notification to a cluster client, in order to inform the client about one power profile supported by the server. The notification contains essential information about the power profile, including the energy phases supported by the profile (and certain details about them). If the server supports multiple power profiles, the function must be called for each profile separately.

On receiving the notification on the client, the event E_CLD_PP_CMD_POWER_PROFILE_NOTIFICATION is generated, containing the sent power profile information in a `tsCLD_PP_PowerProfilePayload` structure (see Section 21.10.4).

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the message. The TSN in the response is 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 local endpoint on which cluster server resides
u8DestinationEndPointId : Number of remote endpoint on which cluster client resides
psDestinationAddress: Pointer to a structure containing the destination address of the client node
pu8TransactionSequenceNumber: Pointer to a location to receive the Transaction Sequence Number (TSN) of the message
psPayload: Pointer to structure containing the payload for the request (see Section 21.10.4), including essential information about the power profile

Returns

E_ZCL_SUCCESS
E_ZCL_FAIL
E_ZCL_ERR_EP_UNKNOWN
E_ZCL_ERR_ZBUFFER_FAIL
E_ZCL_ERR_ZTRANSMIT_FAIL

21.7.2.7 eCLD_PPEnergyPhaseScheduleStateNotificationSend

```
teZCL_Status eCLD_PPEnergyPhasesScheduleStateNotificationSend(
 uint8 u8SourceEndPointId,
 uint8 u8DestinationEndPointId,
```

```
  tsZCL_Address *psDestinationAddress,
  uint8 *pu8TransactionSequenceNumber,
  tsCLD_PP_EnergyPhasesSchedulePayload
  *psPayload);
```

Description

This function can be used on the cluster server to send an Energy Phases Schedule State Notification to a cluster client, in order to inform the client of the energy phases that are in the schedule of a particular power profile. The function is used to send an unsolicited command.

On receiving the notification on the client, the event E_CLD_PP_CMD_ENERGY_PHASES_SCHEDULE_STATE_NOTIFICATION is generated, containing the sent power profile information in a `tsCLD_PP_EnergyPhasesSchedulePayload` structure (see Section 21.10.6).

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the message. The TSN in the response is 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 local endpoint on which cluster server resides
u8DestinationEndPointId: Number of remote endpoint on which cluster client resides
psDestinationAddress: Pointer to a structure containing the destination address of the client node
pu8TransactionSequenceNumber: Pointer to a location to receive the Transaction Sequence Number (TSN) of the message
psPayload: Pointer to structure containing the payload for the request (see Section 21.10.6), including the identifier of the relevant power profile and the associated schedule of energy phases

Returns

E_ZCL_SUCCESS
E_ZCL_FAIL
E_ZCL_ERR_EP_UNKNOWN
E_ZCL_ERR_ZBUFFER_FAIL
E_ZCL_ERR_ZTRANSMIT_FAIL

21.7.2.8 eCLD_PPPowerProfileScheduleConstraintsNotificationSend

```
teZCL_Status eCLD_PPPowerProfileScheduleConstraintsNotificationSend(
 uint8 u8SourceEndPointId,
 uint8 u8DestinationEndPointId,
 tsZCL_Address *psDestinationAddress,
 uint8 *pu8TransactionSequenceNumber,
 tsCLD_PP_PowerProfileScheduleConstraintsPayload
            *psPayload);
```

Description

This function can be used on the cluster server to send a Power Profile Schedule Constraints Notification to a cluster client, in order to inform the client of the schedule restrictions on a particular power profile. The constraints are specified in a `tsCLD_PP_PowerProfileScheduleConstraintsPayload` structure (see Section 21.10.7). They can subsequently be used by the client in calculating the schedule for the energy phases of the power profile. The function is used to send an unsolicited command.

On receiving the notification on the client, the event E_CLD_PP_CMD_SCHEDULE_CONSTRAINTS_NOTIFICATION is generated, containing the sent power profile constraint information in a `tsCLD_PP_PowerProfileScheduleConstraintsPayload` structure.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the message. The TSN in the response is 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 local endpoint on which cluster server resides
u8DestinationEndPointId : Number of remote endpoint on which cluster client resides
psDestinationAddress: Pointer to a structure containing the destination address of the client node
pu8TransactionSequenceNumber: Pointer to a location to receive the Transaction Sequence Number (TSN) of the message
psPayload: Pointer to structure containing the payload for the request (see Section 21.10.7), including the identifier of the relevant power profile and the associated schedule constraints

Returns

E_ZCL_SUCCESS
E_ZCL_FAIL
E_ZCL_ERR_EP_UNKNOWN
E_ZCL_ERR_ZBUFFER_FAIL
E_ZCL_ERR_ZTRANSMIT_FAIL

21.7.2.9 eCLD_PPEnergyPhasesScheduleReqSend

```
teZCL_Status eCLD_PPEnergyPhasesScheduleReqSend(
 uint8 u8SourceEndPointId,
 uint8 u8DestinationEndPointId,
 tsZCL_Address *psDestinationAddress,
 uint8 *pu8TransactionSequenceNumber,
 tsCLD_PP_PowerProfileReqPayload *psPayload);
```

Description

This function can be used on the cluster server to send an Energy Phases Schedule Request to a cluster client, in order to obtain the schedule of energy phases for a particular power profile.

The function is non-blocking and will return immediately. On successfully receiving an Energy Phases Schedule Response from the client, an `E_CLD_PP_CMD_ENERGY_PHASE_SCHEDULE_RSP` event is generated on the server, containing the requested schedule information in a `tsCLD_PP_EnergyPhasesSchedulePayload` structure (see Section 21.10.6). For full details of handling an Energy Phases Schedule Request, refer to Section 21.5.4.1.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the message. The TSN in the response is 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 local endpoint on which cluster server resides:

u8DestinationEndPointId: Number of remote endpoint on which cluster client resides
psDestinationAddress: Pointer to a structure containing the destination address of the client node
pu8TransactionSequenceNumber: Pointer to a location to receive the Transaction Sequence Number (TSN) of the message
psPayload: Pointer to structure containing the payload for the request (see Section 21.10.3), including the identifier of the relevant power profile

Returns

E_ZCL_SUCCESS
E_ZCL_FAIL
E_ZCL_ERR_EP_UNKNOWN
E_ZCL_ERR_ZBUFFER_FAIL
E_ZCL_ERR_ZTRANSMIT_FAIL

21.7.2.10 eCLD_PPPowerProfileStateNotificationSend

```
teZCL_Status eCLD_PPPowerProfileStateNotificationSend(
 uint8 u8SourceEndPointId,
 uint8 u8DestinationEndPointId,
 tsZCL_Address *psDestinationAddress,
 uint8 *pu8TransactionSequenceNumber,
 tsCLD_PP_PowerProfileStatePayload *psPayload);
```

Description

This function can be used on the cluster server to send a Power Profile State Notification to a cluster client, in order to inform the client of the state of the power profile that is currently active on the server. The function is used to send an unsolicited command.

On receiving the notification on the client, the event E_CLD_PP_CMD_POWER_PROFILE_STATE_NOTIFICATION is generated, containing the sent power profile state information in a `tsCLD_PP_PowerProfileStatePayload` structure (see Section 21.10.5).

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the message. The TSN in the response is 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 local endpoint on which cluster server resides
u8DestinationEndPointId: Number of remote endpoint on which cluster client resides
psDestinationAddress: Pointer to a structure containing the destination address of the client node
pu8TransactionSequenceNumber: Pointer to a location to receive the Transaction Sequence Number (TSN) of the message
psPayload: Pointer to structure containing the payload for the command (see Section 21.10.5), including the identifier and state of the relevant power profile

Returns

E_ZCL_SUCCESS
E_ZCL_FAIL

E_ZCL_ERR_EP_UNKNOWN
E_ZCL_ERR_ZBUFFER_FAIL
E_ZCL_ERR_ZTRANSMIT_FAIL

21.7.2.11 eCLD_PPGetPowerProfilePriceSend

```
teZCL_Status eCLD_PPGetPowerProfilePriceSend(
        uint8 u8SourceEndPointId,
        uint8 u8DestinationEndPointId,
        tsZCL_Address *psDestinationAddress,
        uint8 *pu8TransactionSequenceNumber,
        tsCLD_PP_PowerProfileReqPayload *psPayload);
```

Description

This function can be used on the cluster server to send a Get Power Profile Price Request to a cluster client, in order to request the cost of executing the schedule of a particular power profile. Use of this function must be enabled in the cluster compile-time options, as described in Section 21.11.

The function is non-blocking and will return immediately. On successfully receiving a Get Power Profile Price Response from the client, an E_CLD_PP_CMD_GET_POWER_PROFILE_PRICE_RSP event is generated on the server, containing the requested price information in a `tsCLD_PP_GetPowerProfilePriceRspPayload` structure (see Section 21.10.9). For full details of handling a Get Power Profile Price Request, refer to Section 21.5.6.1.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the message. The TSN in the response is 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 local endpoint on which cluster server resides
u8DestinationEndPointId Number of remote endpoint on which cluster client resides
psDestinationAddress Pointer to a structure containing the destination address of the client node
pu8TransactionSequenceNumber Pointer to a location to receive the Transaction Sequence Number (TSN) of the message
psPayload Pointer to structure containing the payload for the request (see Section 21.10.5), including the identifier of the relevant power profile

Returns

E_ZCL_SUCCESS
E_ZCL_FAIL
E_ZCL_ERR_EP_UNKNOWN
E_ZCL_ERR_ZBUFFER_FAIL
E_ZCL_ERR_ZTRANSMIT_FAIL

21.7.2.12 eCLD_PPGetPowerProfilePriceExtendedSend

```
teZCL_Status eCLD_PPGetPowerProfilePriceExtendedSend(
        uint8 u8SourceEndPointId,
        uint8 u8DestinationEndPointId,
```

```
        tsZCL_Address *psDestinationAddress,
        uint8 *pu8TransactionSequenceNumber,
        tsCLD_PP_GetPowerProfilePriceExtendedPayload
 *psPayload);
```

Description

This function can be used on the cluster server to send a Get Power Profile Price Extended Request to a cluster client, in order to request specific cost information about a power profile supported by the server. The cost of executing a power profile can be requested with either scheduled energy phases or contiguous energy phases (no gaps between them). Use of this function must be enabled in the cluster compile-time options, as described in Section 21.11.

The function is non-blocking and will return immediately. On successfully receiving a Get Power Profile Price Extended Response from the client, an E_CLD_PP_CMD_GET_POWER_PROFILE_PRICE_EXTENDED_RSP event is generated on the server, containing the requested price information in a `tsCLD_PP_GetPowerProfilePriceRspPayload` structure (see Section 21.10.9). For full details of handling a Get Power Profile Price Extended Request, refer to Section 21.5.6.1.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the message. The TSN in the response is 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 local endpoint on which cluster server resides
u8DestinationEndPointId Number of remote endpoint on which cluster client resides
psDestinationAddress Pointer to a structure containing the destination address of the client node
pu8TransactionSequenceNumber Pointer to a location to receive the Transaction Sequence Number (TSN) of the message
psPayload Pointer to structure containing the payload for the request (see Section 21.10.8), including the type of price information required

Returns

E_ZCL_SUCCESS
E_ZCL_FAIL
E_ZCL_ERR_EP_UNKNOWN
E_ZCL_ERR_ZBUFFER_FAIL
E_ZCL_ERR_ZTRANSMIT_FAIL

21.7.2.13 eCLD_PPGetOverallSchedulePriceSend

```
teZCL_Status eCLD_PPGetOverallSchedulePriceSend(
    uint8    u8SourceEndPointId,
    uint8 u8DestinationEndPointId,
    tsZCL_Address *psDestinationAddress,
    uint8 *pu8TransactionSequenceNumber);
```

Description

This function can be used on the cluster server to send a Get Overall Schedule Price Request to a cluster client, in order to obtain the overall cost of all the power profiles that are executed over the next 24 hours. Use of this function must be enabled in the cluster compile-time options, as described in Section 21.11.

The function is non-blocking and will return immediately. On successfully receiving a Get Overall Schedule Price Response from the client, an E_CLD_PP_CMD_GET_OVERALL_SCHEDULE_PRICE_RSP event is generated on the server, containing the required price information in a `tsCLD_PP_GetOverallSchedulePriceRspPayload` structure (see Section 21.10.10). For full details of handling a Get Overall Schedule Price Request, refer to Section 21.5.6.2.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the message. The TSN in the response is 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 local endpoint on which cluster server resides
u8DestinationEndPointId: Number of remote endpoint on which cluster client resides
psDestinationAddress: Pointer to a structure containing the destination address of the client node
pu8TransactionSequenceNumber: Pointer to a location to receive the Transaction Sequence Number (TSN) of the message

Returns

E_ZCL_SUCCESS
E_ZCL_FAIL
E_ZCL_ERR_EP_UNKNOWN
E_ZCL_ERR_ZBUFFER_FAIL
E_ZCL_ERR_ZTRANSMIT_FAIL

21.7.3 Client Functions

The following Power Profile cluster functions can be used on a cluster client only:

- eCLD_PPPowerProfileRequestSend
- eCLD_PPEnergyPhasesScheduleNotificationSend
- eCLD_PPPowerProfileStateReqSend
- eCLD_PPEnergyPhasesScheduleStateReqSend
- eCLD_PPPowerProfileScheduleConstraintsReqSend

21.7.3.1 eCLD_PPPowerProfileRequestSend

```
teZCL_Status eCLD_PPPowerProfileRequestSend(
        uint8 u8SourceEndPointId,
        uint8 u8DestinationEndPointId,
        tsZCL_Address *psDestinationAddress,
        uint8 *pu8TransactionSequenceNumber,
        tsCLD_PP_PowerProfileReqPayload *psPayload);
```

Description

This function can be used on a cluster client to send a Power Profile Request to the cluster server, in order to obtain one or more power profiles from the server. The function can be used to request a specific power profile (specified using its identifier) or all the power profiles supported by the server (specified using an identifier of zero).

The function is non-blocking and will return immediately. On receiving the server's response, an E_CLD_PP_CMD_POWER_PROFILE_RSP event is generated on the client, containing a power profile in a `tsCLD_PP_PowerProfilePayload` structure (see Section 21.10.4).

When a particular Power Profile ID is specified but:

- The Power Profile ID is not in the valid range, the server will send a default response of INVALID_VALUE.
- The Power Profile ID is in the valid range but no data corresponding to this ID is available, the server will respond with a default response of NOT_FOUND.

When all power profiles on the server are requested, a response is received for each profile separately and, therefore, the above event is generated for each profile reported.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the message. The TSN in the response is 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 local endpoint on which cluster client resides:
u8DestinationEndPointId : Number of remote endpoint on which cluster server resides
psDestinationAddress: Pointer to a structure containing the destination address of the server node
pu8TransactionSequenceNumber: Pointer to a location to receive the Transaction Sequence Number (TSN) of the message
psPayload: Pointer to structure containing the payload for the request (see Section 21.10.3), including the identifier of the required power profile

Returns

E_ZCL_SUCCESS
E_ZCL_FAIL
E_ZCL_ERR_EP_UNKNOWN
E_ZCL_ERR_ZBUFFER_FAIL
E_ZCL_ERR_ZTRANSMIT_FAIL

21.7.3.2 eCLD_PPEnergyPhasesScheduleNotificationSend

```
teZCL_Status eCLD_PPEnergyPhasesScheduleNotificationSend(
        uint8 u8SourceEndPointId,
        uint8 u8DestinationEndPointId,
        tsZCL_Address *psDestinationAddress,
        uint8 *pu8TransactionSequenceNumber,
        tsCLD_PP_EnergyPhasesSchedulePayload
 *psPayload);
```

Description

This function can be used on a cluster client to send an Energy Phases Schedule Notification to the cluster server, in order to start the schedule of energy phases of a power profile on the server. The function is used to send an unsolicited command and should only be called if the server allows itself to be remotely controlled. The command payload specifies the identifiers of the required energy phases and includes the relative start-times of the phases (see Section 21.10.12).

On receiving the notification on the server, the event E_CLD_PP_CMD_ENERGY_PHASE_SCHEDULE_NOTIFICATION is generated, containing the sent energy phase schedule information in a `tsCLD_PP_EnergyPhasesSchedulePayload` structure (see Section 21.10.6). The subsequent handling of this notification is detailed in Section 21.5.4.2.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the message. The TSN in the response is 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 local endpoint on which cluster client resides:
u8DestinationEndPointId : Number of remote endpoint on which cluster server resides
psDestinationAddress: Pointer to a structure containing the destination address of the server node
pu8TransactionSequenceNumber: Pointer to a location to receive the Transaction Sequence Number (TSN) of the message
psPayload: Pointer to structure containing the payload for the command (see Section 21.10.6), including the scheduled energy phases and start-times

Returns

E_ZCL_SUCCESS
E_ZCL_FAIL
E_ZCL_ERR_EP_UNKNOWN
E_ZCL_ERR_ZBUFFER_FAIL
E_ZCL_ERR_ZTRANSMIT_FAIL

21.7.3.3 eCLD_PPPowerProfileStateReqSend

```
teZCL_Status eCLD_PPPowerProfileStateReqSend(
  uint8   u8SourceEndPointId,
  uint8 u8DestinationEndPointId,
  tsZCL_Address *psDestinationAddress,
  uint8 *pu8TransactionSequenceNumber);
```

Description

This function can be used on a cluster client to send a Power Profile State Request to the cluster server, in order to obtain the identifier(s) of the power profile(s) currently supported on the server.

The function is non-blocking and returns immediately. On receiving the server's response, an E_CLD_PP_CMD_POWER_PROFILE_STATE_RSP event is generated on the client, containing the required identifier(s). The response contains the power profile records of all the supported power profiles on the server in a `tsCLD_PP_PowerProfileStatePayload` structure (see Section 21.10.5).

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the message. The TSN in the response is 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 local endpoint on which cluster client resides:
u8DestinationEndPointId : Number of remote endpoint on which cluster server resides
psDestinationAddress: Pointer to a structure containing the destination address of the server node
pu8TransactionSequenceNumber: Pointer to a location to receive the Transaction Sequence Number (TSN) of the message

Returns

E_ZCL_SUCCESS
E_ZCL_FAIL
E_ZCL_ERR_EP_UNKNOWN
E_ZCL_ERR_ZBUFFER_FAIL
E_ZCL_ERR_ZTRANSMIT_FAIL

21.7.3.4 eCLD_PPEnergyPhasesScheduleStateReqSend

```
teZCL_Status eCLD_PPEnergyPhasesScheduleStateReqSend(
  uint8 u8SourceEndPointId,
  uint8 u8DestinationEndPointId,
  tsZCL_Address *psDestinationAddress,
  uint8 *pu8TransactionSequenceNumber,
tsCLD_PP_PowerProfileReqPayload *psPayload);
```

Description

This function can be used on a cluster client to send an Energy Phases Schedule State Request to the cluster server, in order to obtain the schedule of energy phases for a particular power profile on the server. The obtained schedule can be used to re-align the schedule information on the client with the information on the server - for example, after a reset of the client device.

The function is non-blocking and returns immediately. On receiving the server's response, an E_CLD_PP_CMD_ENERGY_PHASES_SCHEDULE_STATE_RSP event is generated on the client, containing the requested schedule information in a `tsCLD_PP_EnergyPhasesSchedulePayload` structure (see Section 21.10.6).

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the message. The TSN in the response is 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 local endpoint on which cluster client resides
u8DestinationEndPointId : Number of remote endpoint on which cluster server resides
psDestinationAddress: Pointer to a structure containing the destination address of the server node
pu8TransactionSequenceNumber: Pointer to a location to receive the Transaction Sequence Number (TSN) of the message

psPayload: Pointer to structure containing the payload for the request (see Section 21.10.3), including the identifier of the relevant power profile

Returns

E_ZCL_SUCCESS
E_ZCL_FAIL
E_ZCL_ERR_EP_UNKNOWN
E_ZCL_ERR_ZBUFFER_FAIL
E_ZCL_ERR_ZTRANSMIT_FAIL

21.7.3.5 eCLD_PPPowerProfileScheduleConstraintsReqSend

```
teZCL_Status eCLD_PPPowerProfileScheduleConstraintsReqSend(
        uint8 u8SourceEndPointId,
        uint8 u8DestinationEndPointId,
        tsZCL_Address *psDestinationAddress,
        uint8 *pu8TransactionSequenceNumber,
        tsCLD_PP_PowerProfileReqPayload *psPayload);
```

Description

This function can be used on a cluster client to send a Power Profile Schedule Constraints Request to the cluster server, in order to obtain the schedule restrictions on a particular power profile on the server. The obtained constraints can subsequently be used in calculating a schedule of energy phases for the power profile (e.g. before calling **eCLD_PPEnergyPhasesScheduleNotificationSend()**).

The function is non-blocking and will return immediately. The server will send a response to this request and on receiving this response, an E_CLD_PP_CMD_GET_POWER_PROFILE_SCHEDULE_CONSTRAINTS_RSP event is generated on the client, containing the requested schedule constraints in a `tsCLD_PP_PowerProfileScheduleConstraintsPayload` structure (see Section 21.10.7).

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the message. The TSN in the response is 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 local endpoint on which cluster client resides
u8DestinationEndPointId : Number of remote endpoint on which cluster server resides
psDestinationAddress: Pointer to a structure containing the destination address of the server node
pu8TransactionSequenceNumber: Pointer to a location to receive the Transaction Sequence Number (TSN) of the message
psPayload: Pointer to structure containing the payload for the request (see Section 21.10.3), including the identifier of the relevant power profile

Returns

E_ZCL_SUCCESS
E_ZCL_FAIL
E_ZCL_ERR_EP_UNKNOWN
E_ZCL_ERR_ZBUFFER_FAIL

E_ZCL_ERR_ZTRANSMIT_FAIL

21.8 Return codes

The Power Profile cluster functions use the ZCL return codes, listed in Section 7.2.

21.9 Enumerations

21.9.1 'Attribute ID' Enumerations

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

```
typedef enum PACK
{
    E_CLD_PP_ATTR_ID_TOTAL_PROFILE_NUM = 0x0000,
    E_CLD_PP_ATTR_ID_MULTIPLE_SCHEDULE,
    E_CLD_PP_ATTR_ID_ENERGY_FORMATTING,
    E_CLD_PP_ATTR_ID_ENERGY_REMOTE,
    E_CLD_PP_ATTR_ID_SCHEDULE_MODE
} teCLD_PP_Cluster_AttrID;
```

21.9.2 'Power Profile State' Enumerations

The following enumerations represent the possible states of a power profile.

```
typedef enum PACK
{
   E_CLD_PP_STATE_IDLE   = 0x00,
   E_CLD_PP_STATE_PROGRAMMED,
   E_CLD_PP_STATE_RUNNING   = 0x02,
   E_CLD_PP_STATE_PAUSED,
   E_CLD_PP_STATE_WAITING_TO_START,
   E_CLD_PP_STATE_WAITING_PAUSED,
   E_CLD_PP_STATE_ENDED,
} teCLD_PP_PowerProfileState;
```

The above enumerations are described in the table below.

Table 42. 'Power Profile State' Enumerations

Enumeration	Description
E_CLD_PP_STATE_IDLE	Not all parameters of the power profile have yet been defined
E_CLD_PP_STATE_ PROGRAMMED	In the programmed state, as all the parameters of the power pro-file have been defined but there is no schedule or a schedule exists but has not been started
E_CLD_PP_STATE_ RUNNING	The power profile is active and an energy phase is running
E_CLD_PP_STATE_PAUSED	The power profile is active but the current energy phase is paused
E_CLD_PP_STATE_ WAITING_TO_START	The power profile is between two energy phases - one has ended and the next one has not yet started. If the next energy phase is the first energy phase of the schedule, this state indicates that schedule has been started but the first energy has not yet started
E_CLD_PP_STATE_ WAITING_PAUSED	The power profile has been paused while in the 'waiting to start' state (described above)
E_CLD_PP_STATE_ENDED	The power profile schedule has finished

21.9.3 'Server-Generated Command' Enumerations

The following enumerations represent the commands that can be generated by the cluster server.

```
typedef enum PACK
{
 E_CLD_PP_CMD_POWER_PROFILE_NOTIFICATION = 0x00,
 E_CLD_PP_CMD_POWER_PROFILE_RSP,
 E_CLD_PP_CMD_POWER_PROFILE_STATE_RSP,
 E_CLD_PP_CMD_GET_POWER_PROFILE_PRICE,
 E_CLD_PP_CMD_POWER_PROFILE_STATE_NOTIFICATION,
 E_CLD_PP_CMD_GET_OVERALL_SCHEDULE_PRICE,
 E_CLD_PP_CMD_ENERGY_PHASES_SCHEDULE_REQ,
 E_CLD_PP_CMD_ENERGY_PHASES_SCHEDULE_STATE_RSP,
 E_CLD_PP_CMD_ENERGY_PHASES_SCHEDULE_STATE_NOTIFICATION,
 E_CLD_PP_CMD_GET_POWER_PROFILE_SCHEDULE_CONSTRAINTS_NOTIFICATION,
 E_CLD_PP_CMD_GET_POWER_PROFILE_SCHEDULE_CONSTRAINTS_RSP,
 E_CLD_PP_CMD_GET_POWER_PROFILE_PRICE_EXTENDED
 } teCLD_PP_ServerGeneratedCommandID;
```

The above enumerations are used to indicate types of Power Profile cluster events and are described in Section 21.6.

21.9.4 'Server-Received Command' Enumerations

The following enumerations represent the commands that can be received by the cluster server (and are therefore generated on the cluster client).

```
typedef enum PACK
{
  E_CLD_PP_CMD_POWER_PROFILE_REQ = 0x00,
  E_CLD_PP_CMD_POWER_PROFILE_STATE_REQ,
  E_CLD_PP_CMD_GET_POWER_PROFILE_PRICE_RSP,
  E_CLD_PP_CMD_GET_OVERALL_SCHEDULE_PRICE_RSP,
  E_CLD_PP_CMD_ENERGY_PHASES_SCHEDULE_NOTIFICATION,
  E_CLD_PP_CMD_ENERGY_PHASES_SCHEDULE_RSP,
  E_CLD_PP_CMD_POWER_PROFILE_SCHEDULE_CONSTRAINTS_REQ,
  E_CLD_PP_CMD_ENERGY_PHASES_SCHEDULE_STATE_REQ,
  E_CLD_PP_CMD_GET_POWER_PROFILE_PRICE_EXTENDED_RSP
 } teCLD_PP_ServerReceivedCommandID;
```

The above enumerations are used to indicate types of Power Profile cluster events and are described in Section 21.6.

21.10 Structures

21.10.1 tsCLD_PPCallBackMessage

For a Power Profile 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 `tsCLD_PPCallBackMessage` structure:

```
typedef struct
{
    uint8 u8CommandId;
#ifdef PP_CLIENT
```

```
    bool bIsInfoAvailable;
#endif
union
{
    tsCLD_PP_PowerProfileReqPayload *psPowerProfileReqPayload;
    tsCLD_PP_GetPowerProfilePriceExtendedPayload
        *psGetPowerProfilePriceExtendedPayload;
} uReqMessage;
union
{    tsCLD_PP_GetPowerProfilePriceRspPayload *psGetPowerProfilePriceRspPayload;
    tsCLD_PP_GetOverallSchedulePriceRspPayload
            *psGetOverallSchedulePriceRspPayload;
    tsCLD_PP_EnergyPhasesSchedulePayload *psEnergyPhasesSchedulePayload;
    tsCLD_PP_PowerProfileScheduleConstraintsPayload
            *psPowerProfileScheduleConstraintsPayload;
    tsCLD_PP_PowerProfilePayload *psPowerProfilePayload;
    tsCLD_PP_PowerProfileStatePayload *psPowerProfileStatePayload;
}uRespMessage;
} tsCLD_PPCallBackMessage;
```

where:

- `u8CommandId` indicates the type of Power Profile command that has been received, one of:
  - E_CLD_PP_CMD_POWER_PROFILE_REQ
  - E_CLD_PP_CMD_ POWER_PROFILE_STATE_REQ
  - E_CLD_PP_CMD_GET_POWER_PROFILE_PRICE_RSP
  - E_CLD_PP_CMD_GET_POWER_PROFILE_PRICE_EXTENDED_RSP
  - E_CLD_PP_CMD_GET_OVERALL_SCHEDULE_PRICE_RSP
  - E_CLD_PP_CMD_ENERGY_PHASES_SCHEDULE_NOTIFICATION
  - E_CLD_PP_CMD_ENERGY_PHASES_SCHEDULE_RSP
  - E_CLD_PP_CMD_GET_POWER_PROFILE_SCHEDULE_CONSTRAINTS_REQ
  - E_CLD_PP_CMD_ENERGY_PHASES_SCHEDULE_STATE_REQ
  - E_CLD_PP_CMD_POWER_PROFILE_NOTIFICATION
  - E_CLD_PP_CMD_POWER_PROFILE_RSP
  - E_CLD_PP_CMD_POWER_PROFILE_STATE_RSP
  - E_CLD_PP_CMD_POWER_PROFILE_STATE_NOTIFICATION
  - E_CLD_PP_CMD_GET_POWER_PROFILE_PRICE
  - E_CLD_PP_CMD_GET_OVERALL_SCHEDULE_PRICE
  - E_CLD_PP_CMD_ENERGY_PHASES_SCHEDULE_REQ
  - E_CLD_PP_CMD_ENERGY_PHASES_SCHEDULE_STATE_RSP
  - E_CLD_PP_CMD_ENERGY_PHASES_SCHEDULE_STATE_NOTIFICATION
  - E_CLD_PP_CMD_SCHEDULE_CONSTRAINTS_NOTIFICATION
  - E_CLD_PP_CMD_GET_POWER_PROFILE_SCHEDULE_CONSTRAINTS_RSP
  - E_CLD_PP_CMD_GET_POWER_PROFILE_PRICE_EXTENDED
- `bIsInfoAvailable` is a client-only boolean field which indicates whether the appropriate type of information (to which the event relates) is held on the client:
  TRUE if the information type is held on the client, FALSE otherwise
- `uReqMessage` is a union containing the command payload for a request, as one of (depending on the value of `u8CommandId`):
  - `psPowerProfileReqPayload` is a pointer to the payload of a Power Profile Request, a Get Power Profile Schedule Constraints Request, an Energy Phases Schedule Request, an Energy Phases Schedule State Request or a Get Power Profile Price Request (see Section 21.10.5).
  - `psGetPowerProfilePriceExtendedPayload` is a pointer to the payload of a Get Power Profile Price Extended Request (see Section 21.10.8).

- `uRespMessage` is a union containing the command payload for a response or notification, as one of (depending on the value of `u8CommandId`):
  - `psGetPowerProfilePriceRspPayload` is a pointer to the payload of a Get Power Profile Price Response or a Get Power Profile Price Extended Response (see Section 21.10.9)
  - `psGetOverallSchedulePriceRspPayload` is a pointer to the payload of a Get Overall Schedule Price Response (seeSection 21.10.10).
  - `psEnergyPhasesSchedulePayload` is a pointer to the payload of an Energy Phases Schedule Response, an Energy Phases Schedule State Response, an Energy Phases Schedule Notification or an Energy Phases Schedule State Notification (see Section 21.6).
  - `psPowerProfileScheduleConstraintsPayload` is a pointer to the payload of a Power Profile Schedule Constraints Response or a Power Profile Schedule Constraints Notification (see Section 21.10.10).
  - `psPowerProfilePayload` is a pointer to the payload of a Power Profile Response or a Power Profile Notification (see Section 21.10.4).
  - `psPowerProfileStatePayload` is a pointer to the payload of a Power Profile State Response or a Power Profile State Notification (see Section 21.10.5).

Note: *The command payload for each command type is indicated in Table 30 and Table 31 in Section 21.6.*

21.10.2 tsCLD_PPEntry

This structure contains the data for a power profile table entry.

```
typedef struct
{
  zuint8  u8PowerProfileId;
  zuint8  u8NumOfTransferredEnergyPhases;
  zuint8  u8NumOfScheduledEnergyPhases;
  zuint8  u8ActiveEnergyPhaseId;
  tsCLD_PP_EnergyPhaseDelay
       asEnergyPhaseDelay[CLD_PP_NUM_OF_ENERGY_PHASES];
   tsCLD_PP_EnergyPhaseInfo
       asEnergyPhaseInfo[CLD_PP_NUM_OF_ENERGY_PHASES];
   zbool   bPowerProfileRemoteControl;
   zenum8   u8PowerProfileState;
   zuint16  u16StartAfter;
   zuint16  u16StopBefore;
} tsCLD_PPEntry;
```

where:

- `u8PowerProfileId` is the identifier of the power profile in the range 1 to 255 (0 is reserved)
- `u8NumOfTransferredEnergyPhases` is the number of energy phases supported within the power profile
- `u8NumOfScheduledEnergyPhases` is the number of energy phases actually scheduled within the power profile (must be less than or equal to the value of `u8NumOfTransferredEnergyPhases`)
- `u8ActiveEnergyPhaseId` is the identifier of the energy phase that is currently active or will be active next (if currently between energy phases)
- `asEnergyPhaseDelay[]` is an array containing timing information on the scheduled energy phases, where each array element corresponds to one energy phase of the schedule (see Section 21.10.12)
- `asEnergyPhaseInfo[]` is an array containing various information on the supported energy phases, where each array element corresponds to one energy phase of the profile (see Section 21.10.11)
- `bPowerProfileRemoteControl` is a boolean indicating whether the power profile can be remotely controlled: TRUE if it can be remotely controlled, FALSE otherwise (this property directly affects the attribute `bEnergyRemote`)
- `u8PowerProfileState` is a value indicating the current state of the power profile (enumerations are provided - see Section 21.9.2)

- u16StartAfter is the minimum time-delay, in minutes, to be implemented between an instruction to start the power profile schedule and actually starting the schedule
- u16StopBefore is the maximum time-delay, in minutes, to be implemented between an instruction to stop the power profile schedule and actually stopping the schedule

21.10.3 tsCLD_PP_PowerProfileReqPayload

This structure contains the payload of various power profile requests.

```
typedef struct
{
 zuint8   u8PowerProfileId;
}tsCLD_PP_PowerProfileReqPayload;
```

where u8PowerProfileId is the identifier of the power profile of interest.

21.10.4 tsCLD_PP_PowerProfilePayload

This structure contains the payload of a Power Profile Response or of a Power Profile Notification, which reports the details of a power profile.

```
typedef struct
{
   zuint8       u8TotalProfileNum;
   zuint8       u8PowerProfileId;
   zuint8       u8NumOfTransferredPhases;
   tsCLD_PP_EnergyPhaseInfo *psEnergyPhaseInfo;
}tsCLD_PP_PowerProfilePayload;
```

where:

- u8TotalProfileNum is the total number of power profiles supported on the originating device
- u8PowerProfileId is the identifier of the power profile being reported
- u8NumOfTransferredPhases is the number of energy phases supported within the power profile
- psEnergyPhaseInfo is a pointer to a structure or an array of structures (see Section 21.10.11) containing information on the supported energy phases, where each array element corresponds to one energy phase of the profile

21.10.5 tsCLD_PP_PowerProfileStatePayload

This structure contains the payload of a Power Profile State Response or of a Power Profile State Notification.

```
typedef struct
{
  zuint8                    u8PowerProfileCount;
  tsCLD_PP_PowerProfileRecord *psPowerProfileRecord;
}tsCLD_PP_PowerProfileStatePayload;
```

where:

- u8PowerProfileCount is the number of power profiles in the payload
- psPowerProfileRecord is a pointer to one or more power profile records (see Section 21.10.13):
  - For a Power Profile State Notification, it is a pointer to the power profile record of the currently active power profile on the server

– For a Power Profile State Response, it is a pointer to an array of power profile records for all the supported power profiles on the server

21.10.6 tsCLD_PP_EnergyPhasesSchedulePayload

This structure contains the payload of an Energy Phases Schedule Response, of an Energy Phases Schedule State Response or of an Energy Phases Schedule State Notification.

```
typedef struct
{
  zuint8   u8PowerProfileId;
  zuint8   u8NumOfScheduledPhases;
  tsCLD_PP_EnergyPhaseDelay
                   *psEnergyPhaseDelay;
}tsCLD_PP_EnergyPhasesSchedulePayload;
```

where:

- `u8PowerProfileId` is the identifier of the power profile being reported
- `u8NumOfScheduledPhases` is the number of energy phases within the power profile schedule
- `psEnergyPhaseDelay` is a pointer to an array containing timing information on the scheduled energy phases, where each array element corresponds to one energy phase of the schedule (see Section 21.10.12)

21.10.7 tsCLD_PP_PowerProfileScheduleConstraintsPayload

This structure contains the payload of a Power Profile Schedule Constraints Response or of a Power Profile Schedule Constraints Notification, which reports the schedule restrictions on a particular power profile.

```
typedef struct
{
  zuint8  u8PowerProfileId;
  zuint16  u16StartAfter;
  zuint16  u16StopBefore;
} tsCLD_PP_PowerProfileScheduleConstraintsPayload;
```

where:

- `u8PowerProfileId` is the identifier of the power profile being reported
- `u16StartAfter` is the minimum time-delay, in minutes, to be implemented between an instruction to start the power profile schedule and actually starting the schedule
- `u16StopBefore` is the maximum time-delay, in minutes, to be implemented between an instruction to stop the power profile schedule and actually stopping the schedule

21.10.8 tsCLD_PP_GetPowerProfilePriceExtendedPayload

This structure contains the payload of a Get Power Profile Price Extended Request, which requests certain price information relating to a particular power profile.

```
typedef struct
{
    zbmap8   u8Options;
    zuint8   u8PowerProfileId;
    zuint16   u16PowerProfileStartTime;
}tsCLD_PP_GetPowerProfilePriceExtendedPayload;
```

where:

- u8Options is a bitmap indicating the type of request:
  - If bit 0 is set to '1' then the u16PowerProfileStartTime field is used, otherwise it is ignored
  - If bit 1 is set to '0' then an estimated price is required for contiguous energy phases (with no gaps between them); if bit 1 is set '1' then an estimated price is required for the energy phases as scheduled (with any scheduled gaps between them)
- u8PowerProfileId is the identifier of the power profile
- u16PowerProfileStartTime is an optional value (see u8Options above) which indicates the required start-time for execution of the power profile, in minutes, as measured from the current time

21.10.9 tsCLD_PP_GetPowerProfilePriceRspPayload

This structure contains the payload of a Get Power Profile Price Response, which is returned in reply to a Get Power Profile Price Request and a Get Power Profile Price Extended Request.

```
typedef struct
{
     zuint8   u8PowerProfileId;
     zuint16   u16Currency;
     zuint32   u32Price;
     zuint8   u8PriceTrailingDigits;
}tsCLD_PP_GetPowerProfilePriceRspPayload;
```

where:

- u8PowerProfileId is the identifier of the power profile
- u16Currency is a value representing the currency in which the price is quoted
- u32Price is the price as an integer value (without a decimal point)
- u8PriceTrailingDigits specifies the position of the decimal point in the price u32Price, by indicating the number of digits after the decimal point

21.10.10 tsCLD_PP_GetOverallSchedulePriceRspPayload

This structure contains the payload of a Energy Phases Schedule Response, which contains the overall cost of all the power profiles that will be executed over the next 24 hours.

```
typedef struct
{
   zuint16   u16Currency;
   zuint32   u32Price;
   zuint8   u8PriceTrailingDigits;
}tsCLD_PP_GetOverallSchedulePriceRspPayload;
```

where:

- u16Currency is a value representing the currency in which the price is quoted
- u32Price represents the price as an integer value (without a decimal point)
- u8PriceTrailingDigits specifies the position of the decimal point in the price u32Price, by indicating the number of digits after the decimal point

21.10.11 tsCLD_PP_EnergyPhaseInfo

This structure contains various pieces of information about a specific energy phase of a power profile.

```
typedef struct
{
```

```
    zuint8  u8EnergyPhaseId;
    zuint8  u8MacroPhaseId;
    zuint16  u16ExpectedDuration;
    zuint16  u16PeakPower;
    zuint16  u16Energy;
    zuint16  u16MaxActivationDelay;
}tsCLD_PP_EnergyPhaseInfo;
```

where:

- `u8EnergyPhaseId` is the identifier of the energy phase
- `u8MacroPhaseId` is a value that may be used to obtain a name/label for the energy phase for display purposes - for example, it may be the index of an entry in a table of ASCII strings
- `u16ExpectedDuration` is the expected duration of the energy phase, in minutes
- `u16PeakPower` is the estimated peak power of the energy phase, in Watts
- `u16Energy` is the estimated total energy consumption, in Joules, during the energy phase (≤ `u16PeakPower` x `u16ExpectedDuration` x `60`)
- `u16MaxActivationDelay` is the maximum delay, in minutes, between the end of the previous energy phase and the start of this energy phase - special values are as follows: 0x0000 if no delay possible, 0xFFFF if first energy phase

Note: *If `u16MaxActivationDelay` is non-zero, a delayed start-time for the energy phase can be set through the structure `tsCLD_PP_EnergyPhaseDelay` (see Section 21.10.12).*

21.10.12 tsCLD_PP_EnergyPhaseDelay

This structure contains the start-time for a particular energy phase of a power profile.

```
typedef struct
{
     zuint8   u8EnergyPhaseId;
     zuint16  u16ScheduleTime;
}tsCLD_PP_EnergyPhaseDelay;
```

where:

- `u8EnergyPhaseId` is the identifier of the energy phase
- `u16ScheduleTime` is the start-time of the energy phase expressed as a delay, in minutes, from the end of the previous energy phase (for the first energy phase of a power profile schedule, this delay is measured from the start of the schedule)

Note: *A delayed start-time for the energy phase can only be set through this structure if the field `u16MaxActivationDelay` of the structure `tsCLD_PP_EnergyPhaseInfo` for this energy phase is non-zero (see Section 21.10.11).*

21.10.13 tsCLD_PP_PowerProfileRecord

This structure contains information about the current state of a power profile.

```
typedef struct
{
     zuint8   u8PowerProfileId;
     zuint8   u8EnergyPhaseId;
     zbool    bPowerProfileRemoteControl;
     zenum8   u8PowerProfileState;
} tsCLD_PP_PowerProfileRecord;
```

where:

- `u8PowerProfileId` is the identifier of the power profile
- `u8EnergyPhaseId` is the identifier of the currently running energy phase or, if currently between energy phases, the next energy phase to be run
- `bPowerProfileRemoteControl` is a boolean indicating whether the power profile can be remotely controlled (from a client device): TRUE if it can be remotely controlled, FALSE otherwise
- `u8PowerProfileState` is an enumeration indicating the current state of the power profile (see Section 21.9.2)

21.10.14 tsCLD_PPCustomDataStructure

The Power Profile cluster requires extra storage space to be allocated to be used by internal functions. The structure definition for this storage is shown below:

```
typedef struct
{
#ifdef (CLD_PP) && (PP_SERVER)
   bool    bOverrideRunning;
   uint8    u8ActSchPhaseIndex;
   tsCLD_PPEntryasPowerProfileEntry[CLD_PP_NUM_OF_POWER_PROFILES];
#endif
  tsZCL_ReceiveEventAddress    sReceiveEventAddress;
  tsZCL_CallBackEvent      sCustomCallBackEvent;
  tsCLD_PPCallBackMessage    sCallBackMessage;
} tsCLD_PPCustomDataStructure;
```

The fields are for internal use and no knowledge of them is required.

21.11 Compile-time Options

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

To enable the Power Profile cluster in the code to be built, it is necessary to add the following line to the file:

```
#define CLD_PP
```

In addition, to enable the cluster as a client or server, it is also necessary to add one of the following lines to the same file:

```
#define PP_SERVER
#define PP_CLIENT
```

The following options can also be configured at compile-time in the **zcl_options.h** file.

Global Attributes

Add this line to enable the optional Attribute Reporting Status attribute:

```
#define CLD_PP_ATTR_ID_ATTRIBUTE_REPORTING_STATUS
```

Add this line to define the value (n) of the Cluster Revision attribute:

```
#define CLD_PP_CLUSTER_REVISION <n>
```

The default value is 1, which corresponds to the revision of the cluster in the ZCL r6 specification (see Section 2.4).

Enabling 'Get Power Profile Price' Command

The optional 'Get Power Profile Price' command can be enabled by adding:

```
#define CLD_PP_CMD_GET_POWER_PROFILE_PRICE
```

Enabling 'Get Power Profile Price Extended' Command

The optional 'Get Power Profile Price Extended' command can be enabled by adding:

```
#define CLD_PP_CMD_GET_POWER_PROFILE_PRICE_EXTENDED
```

Enable 'Get Overall Schedule Price' Command

The optional 'Get Overall Schedule Price' command can be enabled by adding:

```
#define CLD_PP_CMD_GET_OVERALL_SCHEDULE_PRICE
```

Number of Power Profiles

The number of power profiles on the local device can be defined as `n` by adding:

```
#define CLD_PP_NUM_OF_PROFILES n
```

Maximum Number of Energy Phases

The maximum number of energy phases in a power profile can be defined as `n` by adding:

```
#define CLD_PP_NUM_OF_ENERGY_PHASES n
```

By default, this value is 3.

Disabling APS Acknowledgments for Bound Transmissions

APS acknowledgements for bound transmissions from this cluster can be disabled by adding:

```
#define CLD_PP_BOUND_TX_WITH_APS_ACK_DISABLED
```

22 Diagnostics Cluster

This chapter describes the Diagnostics cluster.

The Diagnostics cluster has a Cluster ID of 0x0B05.

22.1 Overview

The Diagnostics cluster allows the operation of the ZigBee PRO stack to be followed over time. It provides a tool for monitoring the performance of individual network nodes, including the routing of packets through these nodes.

Note: *It is strongly recommended that Diagnostics cluster server attributes are stored in persistent memory to allow performance data to be preserved through a device reset or power interruption.*

To use the functionality of this cluster, you must include the file **Diagnostics.h** in your application and enable the cluster by defining CLD_DIAGNOSTICS in the **zcl_options.h** file.

A Diagnostics cluster instance can act as a client or a server. The inclusion of the client or server software must be pre-defined in the application's compile-time options (in addition, if the cluster is to reside on a custom endpoint then the role of client or server must also be specified when creating the cluster instance).

The compile-time options for the Diagnostics cluster are fully detailed in Section 22.5.

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

- Hardware information
- Stack/Network information

Currently, only three attributes from the Stack/Network Information attribute set are supported (see Section 22.2).

This cluster has no associated events. However, reads and writes of the cluster attributes may give rise to ZCL events (the application is responsible for checking that a written value is within the valid range for the target attribute).

22.2 Diagnostics Structure and Attributes

The structure definition for the Diagnostics cluster is:

```
typedef struct
{
#ifdef DIAGNOSTICS_SERVER
 /* Hardware Information attribute set */
 #ifdef CLD_DIAGNOSTICS_ATTR_ID_NUMBER_OF_RESETS
  uint16 u16NumberOfResets;
 #endif
 #ifdef CLD_DIAGNOSTICS_ATTR_ID_PERSISTENT_MEMORY_WRITES
  uint16 u16PersistentMemoryWrites;
 #endif
 /* Stack/Network Information attribute set */
 #ifdef CLD_DIAGNOSTICS_ATTR_ID_MAC_RX_BCAST
  uint32 u32MacRxBcast;
 #endif
 #ifdef CLD_DIAGNOSTICS_ATTR_ID_MAC_TX_BCAST
  uint32 u32MacTxBcast;
 #endif
 #ifdef CLD_DIAGNOSTICS_ATTR_ID_MAC_RX_UCAST
  uint32 u32MacRxUcast;
 #endif
 #ifdef CLD_DIAGNOSTICS_ATTR_ID_MAC_TX_UCAST
```

```
    uint32 u32MacTxUcast;
  #endif
  #ifdef CLD_DIAGNOSTICS_ATTR_ID_MAC_TX_UCAST_RETRY
    uint16 u16MacTxUcastRetry;
  #endif
  #ifdef CLD_DIAGNOSTICS_ATTR_ID_MAC_TX_UCAST_FAIL
    uint16 u16MacTxUcastFail;
  #endif
  #ifdef CLD_DIAGNOSTICS_ATTR_ID_APS_RX_BCAST
    uint16 u16ApsRxBcast;
  #endif
  #ifdef CLD_DIAGNOSTICS_ATTR_ID_APS_TX_BCAST
    uint16 u16ApsTxBcast;
  #endif
  #ifdef CLD_DIAGNOSTICS_ATTR_ID_APS_RX_UCAST
    uint16 u16ApsRxUcast;
  #endif
  #ifdef CLD_DIAGNOSTICS_ATTR_ID_APS_TX_UCAST_SUCCESS
    uint16 u16ApsTxUcastSuccess;
  #endif
  #ifdef CLD_DIAGNOSTICS_ATTR_ID_APS_TX_UCAST_RETRY
    uint16 u16ApsTxUcastRetry;
  #endif
  #ifdef CLD_DIAGNOSTICS_ATTR_ID_APS_TX_UCAST_FAIL
    uint16 u16ApsTxUcastFail;
  #endif
  #ifdef CLD_DIAGNOSTICS_ATTR_ID_ROUTE_DISC_INITIATED
    uint16 u16RouteDiscInitiated;
  #endif
  #ifdef CLD_DIAGNOSTICS_ATTR_ID_NEIGHBOR_ADDED
    uint16 u16NeighborAdded;
  #endif
  #ifdef CLD_DIAGNOSTICS_ATTR_ID_NEIGHBOR_REMOVED
    uint16 u16NeighborRemoved;
  #endif
  #ifdef CLD_DIAGNOSTICS_ATTR_ID_NEIGHBOR_STALE
    uint16 u16NeighborStale;
  #endif
  #ifdef CLD_DIAGNOSTICS_ATTR_ID_JOIN_INDICATION
    uint16 u16JoinIndication;
  #endif
  #ifdef CLD_DIAGNOSTICS_ATTR_ID_CHILD_MOVED
    uint16 u16ChildMoved;
  #endif
  #ifdef CLD_DIAGNOSTICS_ATTR_ID_NWK_FC_FAILURE
    uint16 u16NWKFCFailure;
  #endif
  #ifdef CLD_DIAGNOSTICS_ATTR_ID_APS_FC_FAILURE
    uint16 u16APSFCFailure;
  #endif
  #ifdef CLD_DIAGNOSTICS_ATTR_ID_APS_UNAUTHORIZED_KEY
    uint16 u16APSUnauthorizedKey;
  #endif
  #ifdef CLD_DIAGNOSTICS_ATTR_ID_NWK_DECRYPT_FAILURE
    uint16 u16NWKDecryptFailure;
  #endif
  #ifdef CLD_DIAGNOSTICS_ATTR_ID_APS_DECRYPT_FAILURE
    uint16 u16APSDecryptFailure;
  #endif
  #ifdef CLD_DIAGNOSTICS_ATTR_ID_PACKET_BUFFER_ALLOCATE_FAILURE
    uint16 u16PacketBufferAllocateFailure;
  #endif
  #ifdef CLD_DIAGNOSTICS_ATTR_ID_RELAYED_UCAST
    uint16 u16RelayedUcast;
  #endif
  #ifdef CLD_DIAGNOSTICS_ATTR_ID_PHY_TO_MAC_QUEUE_LIMIT_REACHED
    uint16 u16PhyToMACQueueLimitReached;
```

```
 #endif
 #ifdef CLD_DIAGNOSTICS_ATTR_ID_PACKET_VALIDATE_DROP_COUNT
  uint16 u16PacketValidateDropCount;
 #endif
 #ifdef CLD_DIAGNOSTICS_ATTR_ID_AVERAGE_MAC_RETRY_PER_APS_MESSAGE_SENT
  uint16 u16AverageMACRetryPerAPSMessageSent;
 #endif
 #ifdef CLD_DIAGNOSTICS_ATTR_ID_LAST_MESSAGE_LQI
  uint8 u8LastMessageLQI;
 #endif
 #ifdef CLD_DIAGNOSTICS_ATTR_ID_LAST_MESSAGE_RSSI
  int8 i8LastMessageRSSI;
 #endif
#endif
 zuint16 u16ClusterRevision;
} tsCLD_Diagnostics;
```

where:

‘Hardware Information’ Attribute Set

The following two attributes can be maintained by the application using the Attribute Access functions detailed in Section 5.2.

- `u16NumberOfResets` is an optional attribute which acts as a counter of device resets/restarts (note that a factory reset clears this attribute) - thus, the attribute value must be incremented on each restart.
- `u16PersistentMemoryWrites` is an optional attribute which acts as a counter of the number of writes to persistent memory - thus, the attribute value must be incremented on each write.

‘Stack/Network Information’ Attribute Set

The following attributes must be updated by the application by calling the function **eCLD_DiagnosticsUpdate()** (see Section 22.3) either periodically (at the highest rate possible) or on receiving an appropriate event from the stack.

```
u32MacRxBcast is reserved for future use
u32MacTxBcast is reserved for future use
u32MacRxUcast is reserved for future use
u32MacTxUcast is reserved for future use
u16MacTxUcastRetry is reserved for future use
u16MacTxUcastFail is reserved for future use
u16ApsRxBcast is reserved for future use
u16ApsTxBcast is reserved for future use
u16ApsRxUcast is reserved for future use
u16ApsTxUcastSuccess is reserved for future use
u16ApsTxUcastRetry is reserved for future use
u16ApsTxUcastFail is reserved for future use
u16RouteDiscInitiated is reserved for future use
u16NeighborAdded is reserved for future use
u16NeighborRemoved is reserved for future use
u16NeighborStale is reserved for future use
u16JoinIndication is reserved for future use
u16ChildMoved is reserved for future use
u16NWKFCFailure is reserved for future use
u16APSFCFailure is reserved for future use
u16APSUnauthorizedKey is reserved for future use
u16NWKDecryptFailure is reserved for future use
u16APSDecryptFailure is reserved for future use
u16PacketBufferAllocateFailure is reserved for future use
u16RelayedUcast is reserved for future use
u16PhyToMACQueueLimitReached is reserved for future use
u16PacketValidateDropCount is reserved for future use
```

- `u16AverageMACRetryPerAPSMessageSent` is an optional attribute which is used to maintain a record of the average number of IEEE802.15.4 MAC-level retries needed to send a message from the APS layer of the stack.
- `u8LastMessageLQI` is an optional attribute containing the LQI (Link Quality Indicator) value for the last message received, as a value in the range 0 to 255 where 0 indicates the worst link quality and 255 indicates the best link quality.
- `i8LastMessageRSSI` is an optional attribute containing the RSSI (Receive Signal Strength Indication) value of the last message received.

Note: *If the value of `u8LastMessageLQI` or `i8LastMessageRSSI` is read remotely, the returned value will relate to the received message that contained the instruction to read the attribute.*

Global Attributes

- `u16ClusterRevision` is a mandatory attribute that specifies the revision of the cluster specification on which this cluster instance is based. The cluster specification in the ZCL r6 corresponds to a cluster revision of 1. The value is incremented by one for each subsequent revision of the cluster specification. This attribute is also described in Section 2.4.

22.3 Functions

The following Diagnostics cluster functions are provided:

- eCLD_DiagnosticsCreateDiagnostics
- eCLD_DiagnosticsUpdate

The cluster attributes can also all be accessed using the general attribute read/write functions, as described in Section 2.3.

22.3.1 eCLD_DiagnosticsCreateDiagnostics

```
teZCL_Status eCLD_DiagnosticsCreateDiagnostics(
  tsZCL_ClusterInstance *psClusterInstance,
  bool_t bIsServer,
  tsZCL_ClusterDefinition *psClusterDefinition,
  void *pvEndPointSharedStructPtr,
  uint8 *pu8AttributeControlBits);
```

Description

This function creates an instance of the Diagnostics 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 creates a Diagnostics cluster instance on the endpoint, but instances of other clusters may also be created on the same endpoint by calling their corresponding creation functions.

Note: *This function must not be called for an endpoint on which a standard ZigBee device is used. In this case, the device and its supported clusters must be registered on the endpoint using the relevant device registration function.*

When used, this function must be called after the stack has been started and after the ZCL has been initialized.

Parameters

- *psClusterInstance*: Pointer to structure containing information about the cluster instance to be created (see Section 6.1.16). This structure is updated by the function by initializing individual structure fields.
- *bIsServer*: Type of cluster instance (server or client) to be created:
  - TRUE - server
  - FALSE - client
- *psClusterDefinition*: Pointer to structure indicating the type of cluster to be created (see Section 6.1.2). In this case, this structure must contain the details of the Diagnostics cluster. This parameter can refer to a pre-filled structure called `sCLD_Diagnostics` which is provided in the **Diagnostics.h** file.
- *pvEndPointSharedStructPtr*: Pointer to the shared structure used for attribute storage. This parameter should be the address of the structure of type `tsCLD_Diagnostics` which defines the attributes of Diagnostics cluster. The function initializes the attributes with default values.
- *pu8AttributeControlBits*: Pointer to an array of **uint8** values, with one element for each attribute in the cluster (see above).

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_INVALID_VALUE

22.3.2 eCLD_DiagnosticsUpdate

```
teZCL_Status eCLD_DiagnosticsUpdate(
 uint8 u8SourceEndPointId);
```

Description

This function updates the (three) Stack/Network Information attributes (see Section 22.2). It should be called periodically by the application (on the cluster server) at the highest rate possible or when an appropriate stack event occurs.

The attributes can otherwise be accessed (for example, read) using the Attribute Access functions detailed in Section 5.2.

Parameters

- *u8SourceEndPointId*: Number of the local endpoint on which cluster server resides

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND

22.4 Enumerations

22.4.1 teCLD_Diagnostics_AttributeId

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

```
typedef enum
{
 /* Hardware Information attribute IDs */
    E_CLD_DIAGNOSTICS_ATTR_ID_NUMBER_OF_RESETS = 0x0000,
    E_CLD_DIAGNOSTICS_ATTR_ID_PERSISTENT_MEMORY_WRITES,
 /* Stack/Network Information attribute IDs */
    E_CLD_DIAGNOSTICS_ATTR_ID_MAC_RX_BCAST = 0x0100,
    E_CLD_DIAGNOSTICS_ATTR_ID_MAC_TX_BCAST,
    E_CLD_DIAGNOSTICS_ATTR_ID_MAC_RX_UCAST,
    E_CLD_DIAGNOSTICS_ATTR_ID_MAC_TX_UCAST,
    E_CLD_DIAGNOSTICS_ATTR_ID_MAC_TX_UCAST_RETRY,
    E_CLD_DIAGNOSTICS_ATTR_ID_MAC_TX_UCAST_FAIL,
    E_CLD_DIAGNOSTICS_ATTR_ID_APS_RX_BCAST,
    E_CLD_DIAGNOSTICS_ATTR_ID_APS_TX_BCAST,
    E_CLD_DIAGNOSTICS_ATTR_ID_APS_RX_UCAST,
    E_CLD_DIAGNOSTICS_ATTR_ID_APS_TX_UCAST_SUCCESS,
    E_CLD_DIAGNOSTICS_ATTR_ID_APS_TX_UCAST_RETRY,
    E_CLD_DIAGNOSTICS_ATTR_ID_APS_TX_UCAST_FAIL,
    E_CLD_DIAGNOSTICS_ATTR_ID_ROUTE_DISC_INITIATED,
    E_CLD_DIAGNOSTICS_ATTR_ID_NEIGHBOR_ADDED,
    E_CLD_DIAGNOSTICS_ATTR_ID_NEIGHBOR_REMOVED,
    E_CLD_DIAGNOSTICS_ATTR_ID_NEIGHBOR_STALE,
    E_CLD_DIAGNOSTICS_ATTR_ID_JOIN_INDICATION,
    E_CLD_DIAGNOSTICS_ATTR_ID_CHILD_MOVED,
    E_CLD_DIAGNOSTICS_ATTR_ID_NWK_FC_FAILURE,
    E_CLD_DIAGNOSTICS_ATTR_ID_APS_FC_FAILURE,
    E_CLD_DIAGNOSTICS_ATTR_ID_APS_UNAUTHORIZED_KEY,
    E_CLD_DIAGNOSTICS_ATTR_ID_NWK_DECRYPT_FAILURE,
    E_CLD_DIAGNOSTICS_ATTR_ID_APS_DECRYPT_FAILURE,
    E_CLD_DIAGNOSTICS_ATTR_ID_PACKET_BUFFER_ALLOCATE_FAILURE,
    E_CLD_DIAGNOSTICS_ATTR_ID_RELAYED_UCAST,
    E_CLD_DIAGNOSTICS_ATTR_ID_PHY_TO_MAC_QUEUE_LIMIT_REACHED,
    E_CLD_DIAGNOSTICS_ATTR_ID_PACKET_VALIDATE_DROP_COUNT,
    E_CLD_DIAGNOSTICS_ATTR_ID_AVERAGE_MAC_RETRY_PER_APS_MESSAGE_SENT,
    E_CLD_DIAGNOSTICS_ATTR_ID_LAST_MESSAGE_LQI,
    E_CLD_DIAGNOSTICS_ATTR_ID_LAST_MESSAGE_RSSI
} teCLD_Diagnostics_AttributeId;
```

22.5 Compile-time Options

To enable the Diagnostics cluster in the code to be built, it is necessary to add the following to the **zcl_options.h** file:

```
#define CLD_DIAGNOSTICS
```

In addition, to include the software for a cluster client or server or both, it is necessary to add one of the following to the same file:

```
#define DIAGNOSTICS_CLIENT
#define DIAGNOSTICS_SERVER
```

Optional Attributes

Add this line to enable the optional Average MAC Retry Per APS Message Sent attribute:

```
#define CLD_DIAGNOSTICS_ATTR_ID_AVERAGE_MAC_RETRY_PER_APS_MESSAGE_SENT
```

Add this line to enable the optional Last Message LQI attribute:

```
#define CLD_DIAGNOSTICS_ATTR_ID_LAST_MESSAGE_LQI
```

Add this line to enable the optional Last Message RSSI attribute:

```
#define CLD_DIAGNOSTICS_ATTR_ID_LAST_MESSAGE_RSSI
```

Global Attributes

Add this line to define the value (n) of the Cluster Revision attribute:

```
#define CLD_DIAGNOSTICS_CLUSTER_REVISION <n>
```

The default value is 1, which corresponds to the revision of the cluster in the ZCL r6 specification (see Section 2.4).

Part IV: Measurement and Sensing Clusters

This part comprises eight chapters:

- Chapter 23 details the **Illuminance Measurement** cluster
- Chapter 24 details the **Illuminance Level Sensing** cluster
- Chapter 25 details the **Temperature Measurement** cluster
- Chapter 26 details the **Pressure Measurement** cluster
- Chapter 27 details the **Flow Measurement** cluster
- Chapter 28 details the **Relative Humidity Measurement** cluster
- Chapter 29 details the **Occupancy Sensing** cluster
- Chapter 30 details the **Electrical Measurement** cluster

23 Illuminance Measurement Cluster

This chapter describes the Illuminance Measurement cluster which is defined in the ZCL and provides an interface to a light sensor that is able to make illuminance measurements.

The Illuminance Measurement cluster has a Cluster ID of 0x0400.

23.1 Overview

The Illuminance Measurement cluster provides an interface to an illuminance measuring device, allowing the configuration of measuring and the reporting of measurements.

To use the functionality of this cluster, you must include the file **IlluminanceMeasurement.h** in your application and enable the cluster by defining CLD_ILLUMINANCE_MEASUREMENT in the **zcl_options.h** file.

An Illuminance Measurement cluster instance can act as a client or a server. The inclusion of the client or server software must be pre-defined in the application's compile-time options (in addition, if the cluster is to reside on a custom endpoint then the role of client or server must also be specified when creating the cluster instance).

The compile-time options for the Illuminance Measurement cluster are fully detailed in Section 23.6.

23.2 Illuminance Measurement Structure and Attributes

The structure definition for the Illuminance Measurement cluster is:

```
typedef struct
{
#ifdef ILLUMINANCE_MEASUREMENT_SERVER
   zuint16    u16MeasuredValue;
   zuint16    u16MinMeasuredValue;
   zuint16    u16MaxMeasuredValue;
#ifdef CLD_ILLMEAS_ATTR_TOLERANCE
  zuint16   u16Tolerance;
#endif
#ifdef CLD_ILLMEAS_ATTR_LIGHT_SENSOR_TYPE
   zenum8   eLightSensorType;
#endif
#ifdef CLD_ILLMEAS_ATTR_ATTRIBUTE_REPORTING_STATUS
   zenum8  u8AttributeReportingStatus;
#endif
#endif
  zuint16  u16ClusterRevision;
} tsCLD_IlluminanceMeasurement;
```

where:

- `u16MeasuredValue` is a mandatory attribute representing the measured illuminance in logarithmic form, calculated as *(10000 x log_{10}Illuminance) + 1*, where the illuminance is measured in Lux (lx). The possible illumination values are in the range 1 lx to 3.576 x 10^6 lx, corresponding to attribute values of 1 to 0xFFFE. The following attribute values have special meaning:
  - 0x0000: Illuminance is too low to be measured.
  - 0xFFFF: Illuminance measurement is invalid.

    The valid range of values of `u16MeasuredValue` can be restricted using the attributes `u16MinMeasuredValue` and `u16MaxMeasuredValue` below - in this case, the attribute can take any value in the range `u16MinMeasuredValue` to `u16MaxMeasuredValue`.

- `u16MinMeasuredValue` is a mandatory attribute representing a lower limit on the value of the attribute `u16MeasuredValue`. The value must be less than that of `u16MaxMeasuredValue`. The value 0xFFFF indicates that the attribute is unused.
- `u16MaxMeasuredValue` is a mandatory attribute representing an upper limit on the value of the attribute `u16MeasuredValue`. The value must be greater than that of `u16MinMeasuredValue`. The value 0xFFFF indicates that the attribute is unused.
- `u16Tolerance` is an optional attribute which indicates the magnitude of the maximum possible error in the value of the attribute `u16MeasuredValue`. The true value is in the range (`u16MeasuredValue` – `u16Tolerance`) to (`u16MeasuredValue` + `u16Tolerance`) .
- `eLightSensorType` is an optional attribute that indicates the type of light sensor to which the cluster is interfaced:
  - 0x00: Photodiode
  - 0x01: CMOS
  - 0x02–0x3F: Reserved
  - 0x40–0xFE: Reserved for manufacturer-specific light sensor types
  - 0xFF: Unknown
- `u8AttributeReportingStatus` is an optional attribute that should be enabled when attribute reporting is used for the cluster (see Section 2.3.5). The value of this attribute indicates whether there are attribute reports still pending (0x00) or the attribute reports are complete (0x01) - all other values are reserved. This attribute is also described in Section 2.4.
- `u16ClusterRevision` is a mandatory attribute that specifies the revision of the cluster specification on which this cluster instance is based. The cluster specification in the ZCL r6 corresponds to a cluster revision of 1. The value is incremented by one for each subsequent revision of the cluster specification. This attribute is also described in Section 2.4.

23.3 Attributes for Default Reporting

The following attributes of the Illuminance Measurement cluster can be selected for default reporting:

```
u16MeasuredValue
u16Tolerance
```

Attribute reporting (including default reporting) is described in Appendix B. Enabling reports for these attributes is described in Appendix B.3.6.

23.4 Functions

The following Illuminance Measurement cluster function is provided in the NXP implementation of the ZCL:

- eCLD_IlluminanceMeasurementCreateIlluminanceMeasurement

The cluster attributes can be accessed using the general attribute read/write functions, as described in Section 2.3.

23.4.1 eCLD_IlluminanceMeasurementCreateIlluminanceMeasurement

```
teZCL_Status eCLD_IlluminanceMeasurementCreateIlluminanceMeasurement(
  tsZCL_ClusterInstance *psClusterInstance,
  bool_t bIsServer,
  tsZCL_ClusterDefinition *psClusterDefinition,
  void *pvEndPointSharedStructPtr,
  uint8 *pu8AttributeControlBits);
```

Description

This function creates an instance of the Illuminance Measurement 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 creates an Illuminance Measurement cluster instance on the endpoint, but instances of other clusters may also be created on the same endpoint by calling their corresponding creation functions.

Note: *This function must not be called for an endpoint on which a standard ZigBee device is used. In this case, the device and its supported clusters must be registered on the endpoint using the relevant device registration function.*

When used, this function must be called after the stack has been started and after the ZCL has been initialized.

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 Illuminance Measurement cluster. The function initializes the array elements to zero.

Parameters

psClusterInstance: Pointer to structure containing information about the cluster instance to be created (see Section 6.1.16). This structure is updated by the function by initializing individual structure fields.
bIsServer : Type of cluster instance (server or client) to be created:
TRUE - server
FALSE - client
psClusterDefinition: Pointer to structure indicating the type of cluster to be created (see Section 6.1.2). In this case, this structure must contain the details of the Illuminance Measurement cluster. This parameter can refer to a pre-filled structure called `sCLD_IlluminanceMeasurement` which is provided in the **IlluminanceMeasurement.h** file.
pvEndPointSharedStructPtr: Pointer to the shared structure used for attribute storage. This parameter should be the address of the structure of type `tsCLD_IlluminanceMeasurement`, which defines the attributes of Illuminance Measurement cluster. The function initializes the attributes with default values.
pu8AttributeControlBits: Pointer to an array of **uint8** values, with one element for each attribute in the cluster (see above).

Returns

E_ZCL_SUCCESS
E_ZCL_FAIL
E_ZCL_ERR_PARAMETER_NULL
E_ZCL_ERR_INVALID_VALUE

23.5 Enumerations

23.5.1 teCLD_IM_ClusterID

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

```
typedef enum
{
```

```
  E_CLD_ILLMEAS_ATTR_ID_MEASURED_VALUE = 0x0000, /* Mandatory */
  E_CLD_ILLMEAS_ATTR_ID_MIN_MEASURED_VALUE, /* Mandatory */
  E_CLD_ILLMEAS_ATTR_ID_MAX_MEASURED_VALUE, /* Mandatory */
  E_CLD_ILLMEAS_ATTR_ID_TOLERANCE,
  E_CLD_ILLMEAS_ATTR_ID_LIGHT_SENSOR_TYPE
} teCLD_IM_ClusterID;
```

23.6 Compile-time options

To enable the Illuminance Measurement cluster in the code to be built, it is necessary to add the following to the **zcl_options.h** file:

```
#define CLD_ILLUMINANCE_MEASUREMENT
```

In addition, to include the software for a cluster client or server, it is necessary to add one of the following to the same file:

```
#define ILLUMINANCE_MEASUREMENT_CLIENT
#define ILLUMINANCE_MEASUREMENT_SERVER
```

Optional Attributes

Add this line to enable the optional Tolerance attribute:

```
#define CLD_ILLMEAS_ATTR_TOLERANCE
```

Add this line to enable the optional Light Sensor Type attribute:

```
#define CLD_ILLMEAS_ATTR_LIGHT_SENSOR_TYPE
```

Global Attributes

Add this line to enable the optional Attribute Reporting Status attribute:

```
#define CLD_ILLMEAS_ATTR_ID_ATTRIBUTE_REPORTING_STATUS
```

Add this line to define the value (n) of the Cluster Revision attribute:

```
#define CLD_ILLMEAS_CLUSTER_REVISION <n>
```

The default value is 1, which corresponds to the revision of the cluster in the ZCL r6 specification (see Section 2.4).

24 Illuminance Level Sensing Cluster

This chapter describes the Illuminance Level Sensing cluster which is defined in the ZCL and provides an interface to light-level sensing functionality.

The Illuminance Level Sensing cluster has a Cluster ID of 0x0401.

24.1 Overview

The Illuminance Level Sensing cluster provides an interface to a device that can sense the local level of illumination. The cluster can configure notifications that are generated when the light-level is above, within or below a certain illuminance band.

To use the functionality of this cluster, you must include the file **IlluminanceLevelSensing.h** in your application and enable the cluster by defining CLD_ILLUMINANCE_LEVEL_SENSING in the **zcl_options.h** file.

An Illuminance Level Sensing cluster instance can act as a client or a server. The inclusion of the client or server software must be pre-defined in the compile-time options of the application. In addition, if the cluster is designed to reside on a custom endpoint, then the role of client or server must also be specified while creating the cluster instance.

The compile-time options for the Illuminance Level Sensing cluster are fully detailed in Section 24.6.

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

- Illuminance Level Sensing Information
- Illuminance Level Sensing Settings

24.2 Cluster structure and attributes

The structure definition for the Illuminance Level Sensing cluster is:

```
typedef struct
{
#ifdef ILLUMINANCE_LEVEL_SENSING_SERVER zenum8 u8LevelStatus;
#ifdef CLD_ILS_ATTR_LIGHT_SENSOR_TYPE zenum8 eLightSensorType;
#endif
 zuint16 u16IlluminanceTargetLevel;
#ifdef CLD_ILS_ATTR_ATTRIBUTE_REPORTING_STATUS
 zenum8 u8AttributeReportingStatus;
#endif
#endif
 zuint16 u16ClusterRevision;
} tsCLD_IlluminanceLevelSensing;
```

where:

Illuminance Level Sensing Information Attributes

- `u8LevelStatus` is a mandatory attribute indicating whether the current illuminance is above, within or below the target band, as follows:

Value	Enumeration	Description
0x00	E_CLD_ILS_LLS_ON_TARGET	Measured illuminance is within the target band
0x01	E_CLD_ILS_LLS_BELOW_TARGET	Measured illuminance is below the target band

Value	Enumeration	Description
0x02	E_CLD_ILS_LLS_ABOVE_TARGET	Measured illuminance is above the target band
0x03 - 0xFF	-	Reserved

- `eLightSensorType` is an optional attribute indicating the type of light-level sensor used, as follows:

Value	Enumeration	Description
0x00	E_CLD_ILS_LST_PHOTODIODE	Photodiode
0x01	E_CLD_ILS_LST_CMOS	CMOS
0x02 - 0x3F	-	Reserved
0x40 - 0xFE	-	Manufacturer-specific types
0xFF	-	Unknown

Illuminance Level Sensing Settings Attribute

- `u16IlluminanceTargetLevel` is a mandatory attribute representing the illuminance level at the centre of the target band. The value of this attribute is calculated as

  $10000 \times \log_{10} Illuminance$

  where *Illuminance* is measured in Lux (lx) and can take values in the range

  $1 \text{ lx} \leq Illuminance \leq 3.576 \times 10^6 \text{ lx}$, corresponding to attribute values in the range 0x0000 to 0xFFFE. The value 0xFFFF is used to indicate that the attribute is invalid.

Note: ***Note 1:*** *The target band is a 'dead band' around the above target level, in which the sensing device is not able to differentiate between different illuminance levels. The width of this band is device-specific.*

Note: ***Note 2:*** *The illuminance status relative to the target band can be monitored by regularly reading the* `u8LevelStatus` *attribute.*

Global Attributes

- `u8AttributeReportingStatus` is an optional attribute that should be enabled when attribute reporting is used for the cluster (see Section 2.3.5). The value of this attribute indicates whether there are attribute reports still pending (0x00) or the attribute reports are complete (0x01) - all other values are reserved. This attribute is also described in Section 2.4.
- `u16ClusterRevision` is a mandatory attribute that specifies the revision of the cluster specification on which this cluster instance is based. The cluster specification in the ZCL r6 corresponds to a cluster revision of 1. The value is incremented by one for each subsequent revision of the cluster specification. This attribute is also described in Section 2.4.

24.3 Attributes for Default Reporting

The following attribute of the Illuminance Level Sensing cluster can be selected for default reporting:

```
u8LevelStatus
```

Attribute reporting (including default reporting) is described in Appendix B. Enabling reports for this attribute is described in Appendix B.3.6.

24.4 Functions

The following Illuminance Level Sensing cluster function is provided in the NXP implementation of the ZCL:

eCLD_IlluminanceLevelSensingCreateIlluminanceLevelSensing

The cluster attributes can be accessed using the general attribute read/write functions, as described in Section 2.3.

24.4.1 eCLD_IlluminanceLevelSensingCreateIlluminanceLevelSensing

```
teZCL_Status eCLD_IlluminanceLevelSensingCreateIlluminanceLevelSensing(
 tsZCL_ClusterInstance *psClusterInstance,
 bool_t bIsServer,
 tsZCL_ClusterDefinition *psClusterDefinition,
 void *pvEndPointSharedStructPtr,
 uint8 *pu8AttributeControlBits);
```

Description

This function creates an instance of the Illuminance Level Sensing 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 creates an Illuminance Level Sensing cluster instance on the endpoint, but instances of other clusters can also be created on the same endpoint by calling their corresponding creation functions.

Note: *This function must not be called for an endpoint on which a standard ZigBee device is used. In this case, the device and its supported clusters must be registered on the endpoint using the relevant device registration function.*

When used, this function must be called after the stack has been started and after the ZCL has been initialized.

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 Illuminance Level Sensing cluster. The function initializes the array elements to zero.

Parameters

- *psClusterInstance*: Pointer to structure containing information about the cluster instance to be created (see Section 6.1.16). This structure is updated by the function by initializing individual structure fields.
- *bIsServer*: Type of cluster instance (server or client) to be created:
  - TRUE - server
  - FALSE - client
- *psClusterDefinition*: Pointer to structure indicating the type of cluster to be created (see Section 6.1.2). In this case, this structure must contain the details of the Illuminance Level Sensing cluster. This parameter can refer to a pre-filled structure called `sCLD_IlluminanceLevelSensing` which is provided in the **IlluminanceLevelSensing.h** file.
- *pvEndPointSharedStructPtr*: Pointer to the shared structure used for attribute storage. This parameter should be the address of the structure of type `tsCLD_IlluminanceLevelSensing`, which defines the attributes of Illuminance Level Sensing cluster. The function initializes the attributes with default values.
- *pu8AttributeControlBits*: Pointer to an array of **uint8** values, with one element for each attribute in the cluster (see above).

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_INVALID_VALUE

24.5 Enumerations

24.5.1 teCLD_ILS_ClusterID

The following structure contains the enumerations used to identify the attributes of the Illuminance Level Sensing cluster (see Section 24.2).

```
typedef enum
{
  E_CLD_ILS_ATTR_ID_LEVEL_STATUS = 0x0000, /* Mandatory */
  E_CLD_ILS_ATTR_ID_LIGHT_SENSOR_TYPE,
  E_CLD_ILS_ATTR_ID_ILLUMINANCE_TARGET_LEVEL = 0x0010, /* Mandatory */
} teCLD_ILS_ClusterID;
```

24.5.2 teCLD_ILS_LightSensorType

The following structure contains the enumerations used to identify the light-level sensor type in the `eLightSensorType` attribute of the cluster (see Section 24.2).

```
typedef enum
{
  E_CLD_ILS_LST_PHOTODIODE = 0,
  E_CLD_ILS_LST_CMOS
} teCLD_ILS_LightSensorType;
```

24.5.3 teCLD_ILS_LightLevelStatus

The following structure contains the enumerations used to represent the light-level status in the `u8LevelStatus` attribute of the cluster (see Section 24.2).

```
typedef enum
{
  E_CLD_ILS_LLS_ON_TARGET,
  E_CLD_ILS_LLS_BELOW_TARGET,
  E_CLD_ILS_LLS_ABOVE_TARGET,
} teCLD_ILS_LightLevelStatus;
```

24.6 Compile-time Options

To enable the Illuminance Level Sensing cluster in the code to be built, it is necessary to add the following to the **zcl_options.h** file:

```
#define CLD_ILLUMINANCE_LEVEL_SENSING
```

In addition, to include the software for a cluster client or server, it is necessary to add one of the following to the same file:

```
#define ILLUMINANCE_LEVEL_SENSING_CLIENT
#define ILLUMINANCE_LEVEL_SENSING_SERVER
```

Optional Attribute

Add this line to enable the optional Light Sensor Type attribute:

```
#define E_CLD_ILS_ATTR_ID_LIGHT_SENSOR_TYPE
```

Global Attributes

Add this line to enable the optional Attribute Reporting Status attribute:

```
#define CLD_ILS_ATTR_ID_ATTRIBUTE_REPORTING_STATUS
```

Add this line to define the value (n) of the Cluster Revision attribute:

```
#define CLD_ILS_CLUSTER_REVISION <n>
```

The default value is 1, which corresponds to the revision of the cluster in the ZCL r6 specification (see Section 2.4).

25 Temperature Measurement Cluster

This chapter describes the Temperature Measurement cluster which is concerned with configuring and reporting temperature measurement.

The Temperature Measurement cluster has a Cluster ID of 0x0402.

25.1 Overview

The Temperature Measurement cluster provides an interface to a temperature measuring device, allowing the configuration of measuring and the reporting of measurements.

To use the functionality of this cluster, you must include the file **TemperatureMeasurement.h** in your application and enable the cluster by defining CLD_TEMPERATURE_MEASUREMENT in the **zcl_options.h** file.

A Temperature Measurement cluster instance can act as a client or a server. The inclusion of the client or server software must be pre-defined in the application's compile-time options (in addition, if the cluster is to reside on a custom endpoint then the role of client or server must also be specified when creating the cluster instance).

The compile-time options for the Temperature Measurement cluster are fully detailed in Section 25.6.

25.2 Temperature Measurement Structure and Attributes

The structure definition for the Temperature Measurement cluster (server) is:

```
typedef struct
{
#ifdef TEMPERATURE_MEASUREMENT_SERVER
  zint16    i16MeasuredValue;
  zint16    i16MinMeasuredValue;
  zint16    i16MaxMeasuredValue;
#ifdef CLD_TEMPMEAS_ATTR_TOLERANCE
  zuint16  u16Tolerance;
#endif
#ifdef CLD_TEMPMEAS_ATTR_ATTRIBUTE_REPORTING_STATUS
  zenum8    u8AttributeReportingStatus;
#endif
#endif
 zuint16 u16ClusterRevision;
} tsCLD_TemperatureMeasurement;
```

where:

- `i16MeasuredValue` is a mandatory attribute representing the measured temperature in degrees Celsius, as follows:
  - `i16MeasuredValue` = 100 x temperature in degrees Celsius
  - The possible values are used as follows:
    - 0x0000 to 0x7FFF represent positive temperatures from 0°C to 327.67ºC.
    - 0x8000 indicates that the temperature measurement is invalid.
    - 0x8001 to 0x954C are unused values.
    - 0x954D to 0xFFFF represent negative temperatures from -273.15°C to -1°C (in two's complement form).
  - This attribute is updated continuously as measurements are made.
- `i16MinMeasuredValue` is a mandatory attribute specifying the value of the attribute i16MeasuredValue which corresponds to the minimum possible temperature that can be measured. Its value must be less than that of the attribute i16MaxMeasuredValue (below). The special value 0x8000 indicates that the minimum is not known.

- `i16MaxMeasuredValue` is a mandatory attribute specifying the value of the attribute i16MeasuredValue which corresponds to the maximum possible temperature that can be measured. Its value must be greater than that of the attribute i16MinMeasuredValue (above). The special value 0x8000 indicates that the maximum is not known.
- `u16Tolerance` is an optional attribute which indicates the magnitude of the maximum possible error in the value of the attribute u16MeasuredValue. The true value is in the range (u16MeasuredValue – u16Tolerance) to (u16MeasuredValue + u16Tolerance).
- `u8AttributeReportingStatus` is an optional attribute that should be enabled when attribute reporting is used for the cluster (see Section 2.3.5). The value of this attribute indicates whether there are attribute reports still pending (0x00) or the attribute reports are complete (0x01) - all other values are reserved. This attribute is also described in Section 2.4.
- `u16ClusterRevision` is a mandatory attribute that specifies the revision of the cluster specification on which this cluster instance is based. The cluster specification in the ZCL r6 corresponds to a cluster revision of 1. The value is incremented by one for each subsequent revision of the cluster specification. This attribute is also described in Section 2.4.

25.3 Attributes for Default Reporting

The following attributes of the Temperature Measurement cluster can be selected for default reporting:

```
i16MeasuredValue
u16Tolerance
```

Attribute reporting (including default reporting) is described in Appendix B. Enabling reports for these attributes is described in Appendix B.3.6.

25.4 Functions

The following Temperature Measurement cluster function is provided in the NXP implementation of the ZCL:eCLD_TemperatureMeasurementCreateTemperatureMeasurement

25.4.1 eCLD_TemperatureMeasurementCreateTemperatureMeasurement

```
teZCL_Status eCLD_TemperatureMeasurementCreateTemperatureMeasurement(
tsZCL_ClusterInstance *psClusterInstance,
  bool_t bIsServer,
  tsZCL_ClusterDefinition *psClusterDefinition,
  void *pvEndPointSharedStructPtr,
  uint8 *pu8AttributeControlBits);
```

Description

This function creates an instance of the Temperature Measurement 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 creates a Temperature Measurement cluster instance on the endpoint, but instances of other clusters may also be created on the same endpoint by calling their corresponding creation functions.

Note: *This function must not be called for an endpoint on which a standard ZigBee device is used. In this case, the device and its supported clusters must be registered on the endpoint using the relevant device registration function.*

When used, this function must be called after the stack has been started and after the ZCL has been initialized.

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 is automatically adjusted by the compiler using the following declaration:

```
uint8 au8TemperatureMeasurementAttributeControlBits
        [(sizeof(asCLD_TemperatureMeasurementClusterAttributeDefinitions)
    /     sizeof(tsZCL_AttributeDefinition))];
```

Parameters

- *psClusterInstance*: Pointer to structure containing information about the cluster instance to be created (see Section 6.1.16). This structure is updated by the function by initializing individual structure fields.
- *bIsServer*: Type of cluster instance (server or client) to be created:
- TRUE - server
- FALSE - client
- *psClusterDefinition*: Pointer to structure indicating the type of cluster to be created (see Section 6.1.2). In this case, this structure must contain the details of the Temperature Measurement cluster. This parameter can refer to a pre-filled structure called `sCLD_TemperatureMeasurement` which is provided in the **TemperatureMeasurement.h** file.
- *pvEndPointSharedStructPtr*: Pointer to the shared structure used for attribute storage. This parameter should be the address of the structure of type `tsCLD_TemperatureMeasurement`which defines the attributes of Temperature Measurement cluster. The function initializes the attributes with default values.
- *pu8AttributeControlBits*: Pointer to an array of **uint8** values, with one element for each attribute in the cluster (see above). For a cluster client, set this pointer to NULL.

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_INVALID_VALUE

25.5 Enumerations

25.5.1 teCLD_TemperatureMeasurement_AttributeID

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

```
typedef enum
{
 E_CLD_TEMPMEAS_ATTR_ID_MEASURED_VALUE = 0x0000, /* Mandatory */
 E_CLD_TEMPMEAS_ATTR_ID_MIN_MEASURED_VALUE,      /* Mandatory */
 E_CLD_TEMPMEAS_ATTR_ID_MAX_MEASURED_VALUE,      /* Mandatory */
 E_CLD_TEMPMEAS_ATTR_ID_TOLERANCE,
} teCLD_TemperatureMeasurement_AttributeID;
```

25.6 Compile-time Options

To enable the Temperature Measurement cluster in the code to be built, it is necessary to add the following to the **zcl_options.h** file:

```
#define CLD_TEMPERATURE_MEASUREMENT
```

In addition, to include the software for a cluster client or server, it is necessary to add one of the following to the same file:

```
#define TEMPERATURE_MEASUREMENT_CLIENT
#define TEMPERATURE_MEASUREMENT_SERVER
```

Optional Attribute

Add this line to enable the optional Tolerance attribute:

```
#define CLD_TEMPMEAS_ATTR_TOLERANCE
```

Global Attributes

Add this line to enable the optional Attribute Reporting Status attribute:

```
#define CLD_TEMPMEAS_ATTR_ID_ATTRIBUTE_REPORTING_STATUS
```

Add this line to define the value (n) of the Cluster Revision attribute:

```
#define CLD_TEMPMEAS_CLUSTER_REVISION <n>
```

The default value is 1, which corresponds to the revision of the cluster in the ZCL r6 specification (see Section 2.4).

26 Pressure Measurement Cluster

This chapter outlines the Pressure Measurement cluster, which provides an interface for configuring and obtaining pressure measurements.

The Pressure Measurement cluster has a Cluster ID of 0x0403.

26.1 Overview

The Pressure Measurement cluster provides an interface for configuring and querying devices that perform pressure measurements.

- The server is located on the device which makes the pressure measurements
- The client is located on another device and queries the server for measurements

The cluster is enabled by defining CLD_PRESSURE_MEASUREMENT in the **zcl_options.h** file. Further compile-time options for the Pressure Measurement cluster are detailed in Section 26.9.

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

- Pressure Measurement Information
- Extended Pressure Measurement Information
- Global

Note that not all of the above attribute sets are currently implemented in the NXP software and not all attributes within a supported attribute set are implemented (see Section 26.2 for the supported attribute sets and attributes).

26.2 Cluster structure and attributes

The structure definition for the Pressure Measurement cluster (server) is:

```
typedef struct
{
#ifdef PRESSURE_MEASUREMENT_SERVER
  zint16 i16MeasuredValue;
  zint16 i16MinMeasuredValue;
  zint16 i16MaxMeasuredValue;
#ifdef CLD_PRESSUREMEAS_ATTR_TOLERANCE
  zuint16 u16Tolerance;
#endif
#ifdef CLD_PRESSUREMEAS_ATTR_ATTRIBUTE_REPORTING_STATUS
  zenum8 u8AttributeReportingStatus;
#endif
#endif
    zuint16 u16ClusterRevision;
} tsCLD_PressureMeasurement;
```

where:

‘Pressure Measurement Information’ Attribute Set

- i16MeasuredValue is a mandatory attribute corresponding to 10 times the measured pressure, in units of kPa, in two's complement form. The range of possible values is 0x8001 (representing -3276.7 kPa) through 0x0000 (0 kPa) to 0x7FFF (representing +3276.7 kPa). The value 0x8000 is used to indicate that the measurement was invalid. In practice, the stored value is limited within the range i16MinMeasuredValue to i16MaxMeasuredValue (see below).

- i16MinMeasuredValue is a mandatory attribute representing a lower limit on the value that can be stored in i16MeasuredValue. It is a two's complement value in the range 0x8001 to 0x7FFE, and it must be less than the value of the attribute i16MaxMeasuredValue.
- i16MaxMeasuredValue is a mandatory attribute representing an upper limit on the value that can be stored in i16MeasuredValue. It is a two's complement value in the range 0x8002 to 0x7FFF, and it must be greater than the value of the attribute i16MinMeasuredValue.

Global Attributes

- `u8AttributeReportingStatus` is an optional attribute that should be enabled when attribute reporting is used for the cluster (see Section 2.3.5). The value of this attribute indicates whether there are attribute reports still pending (0x00) or the attribute reports are complete (0x01) - all other values are reserved. This attribute is also described in Section 2.4.
- `u16ClusterRevision` is a mandatory attribute that specifies the revision of the cluster specification on which this cluster instance is based. The cluster specification in the ZCL r6 corresponds to a cluster revision of 1. The value is incremented by one for each subsequent revision of the cluster specification. This attribute is also described in Section 2.4.

26.3 Initialization and Operation

The Pressure Measurement cluster must be initialized on both the cluster server and client. This can be done using the function **eCLD_PressureMeasurementCreatePressureMeasurement()**, which creates an instance of the Pressure Measurement cluster on a local endpoint.

Once the cluster has been initialized, the application on the server should maintain the cluster attributes (see Section 26.2) with the pressure measurements made by the local device. The application on a client can remotely read these measured values using the ZCL 'Read Attribute' functions, as described in Section 2.3.2.

26.4 Pressure Measurement Events

There are no events specific to the Pressure Measurement cluster.

26.5 Functions

The following Pressure Measurement cluster function is provided:

eCLD_PressureMeasurementCreatePressureMeasurement

26.5.1 eCLD_PressureMeasurementCreatePressureMeasurement

```
teZCL_Status eCLD_PressureMeasurementCreatePressureMeasurement(
 tsZCL_ClusterInstance *psClusterInstance,
 bool_t bIsServer,
 tsZCL_ClusterDefinition *psClusterDefinition,
 void *pvEndPointSharedStructPtr,
 uint8 *pu8AttributeControlBits);
```

Description

This function creates an instance of the Pressure Measurement cluster on an endpoint. The cluster instance is created on the endpoint 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 creates a Pressure Measurement cluster instance on the endpoint, but instances of other clusters can 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 D.

Note: *This function must not be called for an endpoint on which a standard ZigBee device is 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 the ZigBee Devices User Guide (JNUG3131).*

When used, this function must be called after the stack has been started and after the ZCL has been initialized.

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 is automatically adjusted by the compiler using the following declaration:

```
uint8 au8PressureMeasurementAttributeControlBits
[(sizeof(asCLD_PressureMeasurementClusterAttributeDefinitions) /
     sizeof(tsZCL_AttributeDefinition))];
```

Parameters

- *psClusterInstance*: Pointer to structure containing information about the cluster instance to be created (see Section 6.1.16). This structure is updated by the function by initializing individual structure fields.
- *bIsServer*: Type of cluster instance (server or client) to be created:
  - TRUE - server
  - FALSE - client
- *psClusterDefinition*: Pointer to structure indicating the type of cluster to be created (see Section 6.1.2). In this case, this structure must contain the details of the Pressure Measurement cluster. This parameter can refer to a pre-filled structure called `sCLD_PressureMeasurement` which is provided in the **PressureMeasurement.h** file.
- *pvEndPointSharedStructPtr*: Pointer to the shared structure used for attribute storage. This parameter should be the address of the structure of type `tsCLD_PressureMeasurement`, which defines the attributes of Pressure Measurement cluster. The function initializes the attributes with default values.
- *pu8AttributeControlBits*: Pointer to an array of **uint8** values, with one element for each attribute in the cluster (see above). For a cluster client, set this pointer to NULL.

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_INVALID_VALUE

26.6 Return codes

The Pressure Measurement cluster function uses the ZCL return codes defined in Section 7.2.

26.7 Enumerations

26.7.1 'Attribute ID' Enumerations

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

```
typedef enum
{
 E_CLD_PRESSUREMEAS_ATTR_ID_MEASURED_VALUE   = 0x0000,
 E_CLD_PRESSUREMEAS_ATTR_ID_MIN_MEASURED_VALUE,
 E_CLD_PRESSUREMEAS_ATTR_ID_MAX_MEASURED_VALUE,
 E_CLD_PRESSUREMEAS_ATTR_ID_TOLERANCE,
}teCLD_PM_ClusterID;
```

26.8 Structures

There are no structures specific to the Pressure Measurement cluster.

26.9 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 Pressure Measurement cluster.

To enable the Pressure Measurement cluster in the code to be built, it is necessary to add the following line to the file:

```
#define CLD_PRESSURE_MEASUREMENT
```

In addition, to enable the cluster as a client or server, it is also necessary to add one of the following lines to the same file:

```
#define PRESSURE_MEASUREMENT_SERVER
#define PRESSURE_MEASUREMENT_CLIENT
```

The Pressure Measurement cluster contains macros that may be optionally specified at compile-time by adding one or more of the following lines to the **zcl_options.h** file.

Optional Attributes

Add this line to enable the optional Tolerance attribute:

```
#define CLD_PRESSUREMEAS_ATTR_TOLERANCE
```

Global Attributes

Add this line to enable the optional Attribute Reporting Status attribute:

```
#define CLD_PRESSUREMEAS_ATTR_ID_ATTRIBUTE_REPORTING_STATUS
```

Add this line to define the value (n) of the Cluster Revision attribute:

```
#define CLD_PRESSUREMEAS_CLUSTER_REVISION <n>
```

The default value is 1, which corresponds to the revision of the cluster in the ZCL r6 specification (see Section 2.4).

27 Flow Measurement Cluster

This chapter outlines the Flow Measurement cluster, which provides an interface for configuring and obtaining flow measurements for a fluid (liquid or gas).

The Flow Measurement cluster has a Cluster ID of 0x0404.

27.1 Overview

The Flow Measurement cluster provides an interface for configuring and querying devices that perform flow measurements on a fluid (for example, water).

- The server is located on the device which makes the flow measurements
- The client is located on another device and queries the server for measurements

The cluster is enabled by defining CLD_FLOW_MEASUREMENT in the **zcl_options.h** file. Further compile-time options for the Flow Measurement cluster are detailed in Section 27.9.

27.2 Cluster structure and attributes

The structure definition for the Flow Measurement cluster (server) is:

```
typedef struct
{
#ifdef FLOW_MEASUREMENT_SERVER
 zint16  i16MeasuredValue;
 zint16  i16MinMeasuredValue;
 zint16  i16MaxMeasuredValue;
#ifdef CLD_FLOWMEAS_ATTR_TOLERANCE
  zuint16 u16Tolerance;
#endif
#ifdef CLD_FLOWMEAS_ATTR_ATTRIBUTE_REPORTING_STATUS
  zenum8  u8AttributeReportingStatus;
#endif
#endif
 zuint16  u16ClusterRevision;
} tsCLD_FlowMeasurement;
```

where:

‘Flow Measurement Information’ Attribute Set

- `i16MeasuredValue` is a mandatory attribute corresponding to 10 times the measured flow, in units of cubic metres per hour (m3/h). The range of possible values is 0x0000 (0 m3/h) to 0xFFFE (representing 6553.4 m3/h). The value 0xFFFF is used to indicate that the measurement was invalid. In practice, the stored value is limited within the range i16MinMeasuredValue to i16MaxMeasuredValue (see below).
- `i16MinMeasuredValue` is a mandatory attribute representing a lower limit on the value that can be stored in i16MeasuredValue. It is a value in the range 0x0000 to 0x7FFD and, it must be less than the value of the attribute i16MaxMeasuredValue.
- `i16MaxMeasuredValue` is a mandatory attribute representing an upper limit on the value that can be stored in i16MeasuredValue. It is a value in the range 0x0000 to 0x7FFE, and it must be greater than the value of the attribute i16MinMeasuredValue.
- `u16Tolerance` is an optional attribute representing the maximum error associated with the measurement stored in i16MeasuredValue. Thus, the true value of the flow is within the range represented by i16MeasuredValue ± u16Tolerance.

Global Attributes

- `u8AttributeReportingStatus` is an optional attribute that should be enabled when attribute reporting is used for the cluster (see Section 2.3.5). The value of this attribute indicates whether there are attribute reports still pending (0x00) or the attribute reports are complete (0x01) - all other values are reserved. This attribute is also described in Section 2.4.
- `u16ClusterRevision` is a mandatory attribute that specifies the revision of the cluster specification on which this cluster instance is based. The cluster specification in the ZCL r6 corresponds to a cluster revision of 1. The value is incremented by one for each subsequent revision of the cluster specification. This attribute is also described in Section 2.4.

27.3 Initialization and Operation

The Flow Measurement cluster must be initialized on both the cluster server and client. Initialization can be done using the **eCLD_FlowMeasurementCreateFlowMeasurement()** function, which creates an instance of the Flow Measurement cluster on a local endpoint.

Once the cluster has been initialized, the application on the server should maintain the cluster attributes (see Section 27.2) with the flow measurements made by the local device. The application on a client can remotely read these measured values using the ZCL 'Read Attribute' functions, as described in Section 2.3.2.

27.4 Flow Measurement Events

There are no events specific to the Flow Measurement cluster.

27.5 Functions

The following Flow Measurement cluster function is provided:

eCLD_FlowMeasurementCreateFlowMeasurement

27.5.1 eCLD_FlowMeasurementCreateFlowMeasurement

```
teZCL_Status eCLD_FlowMeasurementCreateFlowMeasurement(
  tsZCL_ClusterInstance *psClusterInstance,
  bool_t bIsServer,
  tsZCL_ClusterDefinition *psClusterDefinition,
  void *pvEndPointSharedStructPtr,
  uint8 *pu8AttributeControlBits);
```

Description

This function creates an instance of the Flow Measurement 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 creates a Flow Measurement 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 D.

Note: *This function must not be called for an endpoint on which a standard ZigBee device is 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 the ZigBee Devices User Guide (JNUG3131).*

When used, this function must be called after the stack has been started and after the ZCL has been initialized.

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 is automatically adjusted by the compiler using the following declaration:

```
uint8 au8FlowMeasurementAttributeControlBits
      [(sizeof(asCLD_FlowMeasurementClusterAttributeDefinitions) /
      sizeof(tsZCL_AttributeDefinition))];
```

Parameters

- *psClusterInstance*: Pointer to structure containing information about the cluster instance to be created (see Section 6.1.16). This structure is updated by the function by initializing individual structure fields.
- *bIsServer*: Type of cluster instance (server or client) to be created:
  - TRUE - server
  - FALSE - client
- *psClusterDefinition*: Pointer to structure indicating the type of cluster to be created (see Section 6.1.2). In this case, this structure must contain the details of the Flow Measurement cluster. This parameter can refer to a pre-filled structure called `sCLD_FlowMeasurement` which is provided in the **FlowMeasurement.h** file.
- *pvEndPointSharedStructPtr*: Pointer to the shared structure used for attribute storage. This parameter should be the address of the structure of type `tsCLD_FlowMeasurement`, which defines the attributes of Flow Measurement cluster. The function initializes the attributes with default values.
- *pu8AttributeControlBits*: Pointer to an array of **uint8** values, with one element for each attribute in the cluster (see above). For a cluster client, set this pointer to NULL.

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_INVALID_VALUE

27.6 Return codes

The Flow Measurement cluster function uses the ZCL return codes defined in Section 7.2.

27.7 Enumerations

27.7.1 'Attribute ID' Enumerations

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

```
typedef enum
{
    E_CLD_FLOWMEAS_ATTR_ID_MEASURED_VALUE  = 0x0000,
    E_CLD_FLOWMEAS_ATTR_ID_MIN_MEASURED_VALUE,
    E_CLD_FLOWMEAS_ATTR_ID_MAX_MEASURED_VALUE,
```

```
    E_CLD_FLOWMEAS_ATTR_ID_TOLERANCE,
} teCLD_FlowMeasurement_AttributeID;
```

27.8 Structures

There are no structures specific to the Flow Measurement cluster.

27.9 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 Flow Measurement cluster.

To enable the Flow Measurement cluster in the code to be built, it is necessary to add the following line to the file:

```
#define CLD_FLOW_MEASUREMENT
```

In addition, to enable the cluster as a client or server, it is also necessary to add one of the following lines to the same file:

```
#define FLOW_MEASUREMENT_SERVER
#define FLOW_MEASUREMENT_CLIENT
```

The Flow Measurement cluster contains macros that may be optionally specified at compile-time by adding one or more of the following lines to the **zcl_options.h** file.

Optional Attributes

Add this line to enable the optional Tolerance attribute:

```
#define CLD_FLOWMEAS_ATTR_TOLERANCE
```

Global Attributes

Add this line to enable the optional Attribute Reporting Status attribute:

```
#define CLD_FLOWMEAS_ATTR_ID_ATTRIBUTE_REPORTING_STATUS
```

Add this line to define the value (n) of the Cluster Revision attribute:

```
#define CLD_FLOWMEAS_CLUSTER_REVISION <n>
```

The default value is 1, which corresponds to the revision of the cluster in the ZCL r6 specification (see Section 2.4).

28 Relative Humidity Measurement Cluster

This chapter describes the Relative Humidity Measurement cluster which describes how to configure and report relative humidity measurement.

The Relative Humidity Measurement cluster has a Cluster ID of 0x0405.

28.1 Overview

The Relative Humidity Measurement cluster provides an interface to a humidity measuring device, allowing the configuration of relative humidity measuring and the reporting of measurements.

To use the functionality of this cluster, you must include the file **RelativeHumidityMeasurement.h** in your application and enable the cluster by defining CLD_RELATIVE_HUMIDITY_MEASUREMENT in the **zcl_options.h** file.

A Relative Humidity Measurement cluster instance can act as a client or a server. The inclusion of the client or server software must be pre-defined in the application's compile-time options (in addition, if the cluster is to reside on a custom endpoint then the role of client or server must also be specified when creating the cluster instance).

The compile-time options for the Relative Humidity Measurement cluster are fully detailed in Section 28.6.

28.2 RH Measurement Structure and Attributes

The structure definition for the Relative Humidity Measurement cluster (server) is:

```
typedef struct
{
    zuint16    u16MeasuredValue;
    zuint16    u16MinMeasuredValue;
    zuint16    u16MaxMeasuredValue;
#ifdef E_CLD_RHMEAS_ATTR_TOLERANCE
    zuint16    u16Tolerance;
#endif
} tsCLD_RelativeHumidityMeasurement;
```

where:

- `u16MeasuredValue` is a mandatory attribute representing the measured relatively humidity as a percentage in steps of 0.01%, as follows:
  - `u16MeasuredValue` = 100 x relative humidity percentage
  - So, for example, 0x197C represents a relative humidity measurement of 65.24%. The possible values are used as follows:
  - 0x0000 to 0x2710 represent relative humidities from 0% to 100%
  - 0x2711 to 0xFFFE are unused values
  - 0xFFFF indicates an invalid measurement
  - This attribute is updated continuously as measurements are made.
- `u16MinMeasuredValue` is a mandatory attribute specifying the value of the attribute u16MeasuredValue which corresponds to the minimum possible relative humidity that can be measured. Its value must be less than that of the attribute u16MaxMeasuredValue (below). The special value 0xFFFF is used to indicate that the minimum is not defined.
- `u16MaxMeasuredValue` is a mandatory attribute specifying the value of the attribute u16MeasuredValue which corresponds to the maximum possible relative humidity that can be measured. Its value must be greater

than that of the attribute u16MinMeasuredValue (above). The special value 0xFFFF is used to indicate that the maximum is not defined.

- `u16Tolerance` is an optional attribute which indicates the magnitude of the maximum possible error in the value of the attribute u16MeasuredValue. The true value will be in the range (u16MeasuredValue – u16Tolerance) to (u16MeasuredValue + u16Tolerance).
- `u8AttributeReportingStatus` is an optional attribute that should be enabled when attribute reporting is used for the cluster (see Section 2.3.5). The value of this attribute indicates whether there are attribute reports still pending (0x00) or the attribute reports are complete (0x01) - all other values are reserved. This attribute is also described in Section 2.4.
- `u16ClusterRevision` is a mandatory attribute that specifies the revision of the cluster specification on which this cluster instance is based. The cluster specification in the ZCL r6 corresponds to a cluster revision of 1. The value is incremented by one for each subsequent revision of the cluster specification. This attribute is also described in Section 2.4.

28.3 Attributes for Default Reporting

The following attributes of the Relative Humidity Measurement cluster can be selected for default reporting:

```
u16MeasuredValue
u16Tolerance
```

Attribute reporting (including default reporting) is described in Appendix B. Enabling reports for these attributes is described in Appendix B.3.6.

28.4 Functions

The following Relative Humidity Measurement cluster function is provided in the NXP implementation of the ZCL:

eCLD_RelativeHumidityMeasurementCreateRelativeHumidityMeasurement

28.4.1 eCLD_RelativeHumidityMeasurementCreateRelativeHumidityMeasurement

```
teZCL_Status eCLD_RelativeHumidityMeasurementCreateRelativeHumidityMeasurement(
  tsZCL_ClusterInstance *psClusterInstance,
  bool_t bIsServer,
  tsZCL_ClusterDefinition *psClusterDefinition,
  void *pvEndPointSharedStructPtr,
  uint8 *pu8AttributeControlBits);
```

Description

This function creates an instance of the Relative Humidity Measurement cluster on an endpoint. The cluster instance is created on the endpoint 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 creates a Relative Humidity Measurement cluster instance on the endpoint, but instances of other clusters may also be created on the same endpoint by calling their corresponding creation functions.

Note: *This function must not be called for an endpoint on which a standard ZigBee device is used. In this case, the device and its supported clusters must be registered on the endpoint using the relevant device registration function.*

When used, this function must be called after the stack has been started and after the ZCL has been initialized.

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 is automatically adjusted by the compiler using the following declaration:

```
uint8 au8RelativeHumidityMeasurementAttributeControlBits
   [(sizeof(asCLD_RelativeHumidityMeasurementClusterAttributeDefinitions) /
   sizeof(tsZCL_AttributeDefinition))];
```

Parameters

- *psClusterInstance*: Pointer to structure containing information about the cluster instance to be created (see Section 6.1.16). This structure is updated by the function by initializing individual structure fields.
- *bIsServer*: Type of cluster instance (server or client) to be created:
  - TRUE - server
  - FALSE - client
- *psClusterDefinition*: Pointer to structure indicating the type of cluster to be created (see Section 6.1.2). In this case, this structure must contain the details of the Relative Humidity Measurement cluster. This parameter can refer to a pre-filled structure called `sCLD_RelativeHumidityMeasurement` which is provided in the **RelativeHumidityMeasurement.h** file.
- *pvEndPointSharedStructPtr*: Pointer to the shared structure used for attribute storage. This parameter should be the address of the structure of type `tsCLD_RelativeHumidityMeasurement` which defines the attributes of Relative Humidity Measurement cluster. The function initializes the attributes with default values.
- *pu8AttributeControlBits*: Pointer to an array of **uint8** values, with one element for each attribute in the cluster (see above). For a cluster client, set this pointer to NULL.

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_INVALID_VALUE

28.5 Enumerations

28.5.1 teCLD_RHM_ClusterID

The following structure contains the enumerations used to identify the attributes of the Relative Humidity Measurement cluster.

```
typedef enum
{
    E_CLD_RHMEAS_ATTR_ID_MEASURED_VALUE = 0x0000,  /* Mandatory */
    E_CLD_RHMEAS_ATTR_ID_MIN_MEASURED_VALUE,    /* Mandatory */
    E_CLD_RHMEAS_ATTR_ID_MAX_MEASURED_VALUE,    /* Mandatory */
    E_CLD_RHMEAS_ATTR_ID_TOLERANCE,
} teCLD_RHM_ClusterID;
```

28.6 Compile-time Options

To enable the Relative Humidity Measurement cluster in the code to be built, it is necessary to add the following to the **zcl_options.h** file:

```
#define CLD_RELATIVE_HUMIDITY_MEASUREMENT
```

In addition, to include the software for a cluster client or server, it is necessary to add one of the following to the same file:

```
#define RELATIVE_HUMIDITY_MEASUREMENT_CLIENT
#define RELATIVE_HUMIDITY_MEASUREMENT_SERVER
```

Optional Attribute

Add this line to enable the optional Tolerance attribute:

```
#define CLD_RHMEAS_ATTR_TOLERANCE
```

Global Attributes

Add this line to enable the optional Attribute Reporting Status attribute:

```
#define CLD_RHMEAS_ATTR_ID_ATTRIBUTE_REPORTING_STATUS
```

Add this line to define the value (n) of the Cluster Revision attribute:

```
#define CLD_RHMEAS_CLUSTER_REVISION <n>
```

The default value is 1, which corresponds to the revision of the cluster in the ZCL r6 specification (see Section 2.4).

29 Occupancy Sensing Cluster

This chapter describes the Occupancy Sensing cluster which provides an interface to an occupancy sensor.

The Occupancy Sensing cluster has a Cluster ID of 0x0406.

29.1 Overview

The Occupancy Sensing cluster provides an interface to an occupany sensor, allowing the configuration of sensing and the reporting of status.

To use the functionality of this cluster, you must include the file **OccupancySensing.h** in your application and enable the cluster by defining CLD_OCCUPANCY_SENSING in the **zcl_options.h** file.

An Occupancy Sensing cluster instance can act as a client or a server. The inclusion of the client or server software must be pre-defined in the application's compile-time options (in addition, if the cluster is to reside on a custom endpoint then the role of client or server must also be specified when creating the cluster instance).

The compile-time options for the Occupancy Sensing cluster are fully detailed in Section 29.6.

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

- Occupancy sensor information
- PIR configuration
- Ultrasonic configuration

This cluster has no associated events. The status of an occupancy sensor can be obtained by reading the 'occupancy' attribute (see Section 29.2) which is automatically maintained by the cluster server. The cluster attributes can be accessed using the general attribute read/write functions, as described in Section 2.3.

29.2 Occupancy Sensing Structure and Attributes

The structure definition for the Occupancy Sensing cluster is:

```
typedef struct
{
#ifdef OCCUPANCY_SENSING_SERVER
    zbmap8   u8Occupancy;
    zenum8   eOccupancySensorType;
#ifdef CLD_OS_ATTR_PIR_OCCUPIED_TO_UNOCCUPIED_DELAY
    zuint16 u16PIROccupiedToUnoccupiedDelay;
#endif
#ifdef CLD_OS_ATTR_PIR_UNOCCUPIED_TO_OCCUPIED_DELAY
    zuint8  u8PIRUnoccupiedToOccupiedDelay;
#endif
#ifdef CLD_OS_ATTR_PIR_UNOCCUPIED_TO_OCCUPIED_THRESHOLD
    zuint8  u8PIRUnoccupiedToOccupiedThreshold;
#endif
#ifdef CLD_OS_ATTR_ULTRASONIC_OCCUPIED_TO_UNOCCUPIED_DELAY
    zuint16 u16UltrasonicOccupiedToUnoccupiedDelay;
#endif
#ifdef CLD_OS_ATTR_ULTRASONIC_UNOCCUPIED_TO_OCCUPIED_DELAY
    zuint8  u8UltrasonicUnoccupiedToOccupiedDelay;
#endif
#ifdef CLD_OS_ATTR_ULTRASONIC_UNOCCUPIED_TO_OCCUPIED_THRESHOLD
    zuint8      u8UltrasonicUnoccupiedToOccupiedThreshold;
#endif
#ifdef CLD_OS_ATTR_ATTRIBUTE_REPORTING_STATUS
    zenum8  u8AttributeReportingStatus;
```

```
#endif
#endif
    zuint16 u16ClusterRevision;
} tsCLD_OccupancySensing;
```

where:

‘Occupancy Sensor Information’ Attribute Set

- `u8Occupancy` is a mandatory attribute indicating the sensed occupancy in a bitmap in which bit 0 is used as follows (and all other bits are reserved):
  - bit 0 = 1 : occupied
  - bit 0 = 0 : unoccupied
- `eOccupancySensorType` is a mandatory attribute indicating the type of occupancy sensor, as follows:
  - 0x00 : PIR
  - 0x01 : Ultrasonic
  - 0x02 : PIR and ultrasonic

‘PIR Configuration’ Attribute Set

- `u16PIROccupiedToUnoccupiedDelay` is an optional attribute for a PIR detector representing the time delay, in seconds, between the last detected movement and the sensor changing its occupancy state from ‘occupied’ to ‘unoccupied’
- `u8PIRUnoccupiedToOccupiedDelay` is an optional attribute for a PIR detector representing the time delay, in seconds, between the detection of movement and the sensor changing its occupancy state from ‘unoccupied’ to ‘occupied’. The interpretation of this attribute changes when it is used in conjunction with the corresponding threshold attribute (see below)
- `u8PIRUnoccupiedToOccupiedThreshold` is an optional threshold attribute that can be used in conjunction with the delay attribute `u8PIRUnoccupiedToOccupiedDelay` to allow for false positive detections. Use of this threshold attribute changes the interpretation of the delay attribute. The threshold represents the minimum number of detections required within the delay-period before the sensor will change its occupancy state from ‘unoccupied’ to ‘occupied’. The minimum valid threshold value is 1

‘Ultrasonic Configuration’ Attribute Set

- `u16UltrasonicOccupiedToUnoccupiedDelay` is an optional attribute for an Ultrasonic detector representing the time delay, in seconds, between the last detected movement and the sensor changing its occupancy state from ‘occupied’ to ‘unoccupied’
- `u8UltrasonicUnoccupiedToOccupiedDelay` is an optional attribute representing the time delay, in seconds, between the detection of movement and the sensor changing its occupancy state from ‘unoccupied’ to ‘occupied’. The interpretation of this attribute changes when it is used in conjunction with the corresponding threshold attribute (see below)
- `u8UltrasonicUnoccupiedToOccupiedThreshold` is an optional threshold attribute that can be used in conjunction with the delay attribute `u8UltrasonicUnoccupiedToOccupiedDelay` to allow for false positive detections. Use of this threshold attribute changes the interpretation of the delay attribute. The threshold represents the minimum number of detections required within the delay-period before the sensor will change its occupancy state from ‘unoccupied’ to ‘occupied’. The minimum valid threshold value is 1

Note: *The 'Occupied To Unoccupied’ and 'Unoccupied To Occupied' attributes can be used to reduce sensor 'chatter' when an occupancy sensor is deployed in an area in which the occupation frequently changes*

Note: *(e.g. in a corridor).*

Global Attributes

- `u8AttributeReportingStatus` is an optional attribute that should be enabled when attribute reporting is used for the cluster (see Section 2.3.5). The value of this attribute indicates whether there are attribute reports still pending (0x00) or the attribute reports are complete (0x01) - all other values are reserved. This attribute is also described in Section 2.4.

```
u16ClusterRevision is a mandatory attribute that specifies the revision of the cluster spe
```

29.3 Attributes for Default Reporting

The following attribute of the Occupancy Sensing cluster can be selected for default reporting:

```
u8Occupancy
```

Attribute reporting (including default reporting) is described in Appendix B. Enabling reports for this attribute is described in Appendix B.3.6.

29.4 Functions

The following Occupancy Sensing cluster function is provided in the NXP implementation of the ZCL:

eCLD_OccupancySensingCreateOccupancySensing

The cluster attributes can be accessed using the general attribute read/write functions, as described in Section 2.3. The state of the occupancy sensor can be obtained by reading the `u8Occupancy` attribute in the `tsCLD_OccupancySensing` structure on the cluster server (see Section 29.2).

29.4.1 eCLD_OccupancySensingCreateOccupancySensing

```
teZCL_Status eCLD_OccupancySensingCreateOccupancySensing(
  tsZCL_ClusterInstance *psClusterInstance,
  bool_t bIsServer,
  tsZCL_ClusterDefinition *psClusterDefinition,
  void *pvEndPointSharedStructPtr,
  uint8 *pu8AttributeControlBits);
```

Description

This function creates an instance of the Occupancy Sensing 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 creates an Occupancy Sensing cluster instance on the endpoint, but instances of other clusters may also be created on the same endpoint by calling their corresponding creation functions.

Note: *This function must not be called for an endpoint on which a standard ZigBee device is used. In this case, the device and its supported clusters must be registered on the endpoint using the relevant device registration function.*

When used, this function must be called after the stack has been started and after the ZCL has been initialized.

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 Occupancy Sensing cluster. The function initializes the array elements to zero.

Parameters

- *psClusterInstance*: Pointer to structure containing information about the cluster instance to be created (see Section 6.1.16). This structure is updated by the function by initialising individual structure fields.
- *bIsServer* : Type of cluster instance (server or client) to be created:
  - TRUE - server
  - FALSE - client
- *psClusterDefinition*: Pointer to structure indicating the type of cluster to be created (see Section 6.1.2). In this case, this structure must contain the details of the Occupancy Sensing cluster. This parameter can refer to a pre-filled structure called `sCLD_OccupancySensing` which is provided in the **OccupancySensing.h** file.
- *pvEndPointSharedStructPtr*: Pointer to the shared structure used for attribute storage. This parameter should be the address of the structure of type `tsCLD_OccupancySensing` which defines the attributes of Occupancy Sensing cluster. The function initializes the attributes with default values.
- *pu8AttributeControlBits*: Pointer to an array of **uint8** values, with one element for each attribute in the cluster (see above).

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_INVALID_VALUE

29.5 Enumerations

29.5.1 teCLD_OS_ClusterID

The following structure contains the enumeration used to identify the attributes of the Occupancy Sensing cluster.

```
typedef enum
{
  E_CLD_OS_ATTR_ID_OCCUPANCY = 0x0000,  /* Mandatory */
  E_CLD_OS_ATTR_ID_OCCUPANCY_SENSOR_TYPE,  /* Mandatory */
  E_CLD_OS_ATTR_ID_PIR_OCCUPIED_TO_UNOCCUPIED_DELAY = 0x0010,
  E_CLD_OS_ATTR_ID_PIR_UNOCCUPIED_TO_OCCUPIED_DELAY,
  E_CLD_OS_ATTR_ID_PIR_UNOCCUPIED_TO_OCCUPIED_THRESHOLD,
  E_CLD_OS_ATTR_ID_ULTRASONIC_OCCUPIED_TO_UNOCCUPIED_DELAY = 0x0020,
  E_CLD_OS_ATTR_ID_ULTRASONIC_UNOCCUPIED_TO_OCCUPIED_DELAY,
  E_CLD_OS_ATTR_ID_ULTRASONIC_UNOCCUPIED_TO_OCCUPIED_THRESHOLD
} teCLD_OS_ClusterID;
```

29.6 Compile-time options

To enable the Occupancy Sensing cluster in the code to be built, it is necessary to add the following to the **zcl_options.h** file:

```
#define CLD_OCCUPANCY_SENSING
```

In addition, to include the software for a cluster client or server or both, it is necessary to add one of the following to the same file:

```
#define OCCUPANCY_SENSING_CLIENT
#define OCCUPANCY_SENSING_SERVER
```

Optional Attributes

Add this line to enable the optional PIR Occupied To Unoccupied Delay attribute:

```
#define CLD_OS_ATTR_PIR_OCCUPIED_TO_UNOCCUPIED_DELAY
```

Add this line to enable the optional PIR Unoccupied To Occupied Delay attribute:

```
#define CLD_OS_ATTR_PIR_UNOCCUPIED_TO_OCCUPIED_DELAY
```

Add this line to enable the optional PIR Unoccupied To Occupied Threshold attribute:

```
#define CLD_OS_ATTR_PIR_UNOCCUPIED_TO_OCCUPIED_THRESHOLD
```

Add this line to enable the optional Ultrasonic Occupied To Unoccupied Delay attribute:

```
#define CLD_OS_ATTR_ULTRASONIC_OCCUPIED_TO_UNOCCUPIED_DELAY
```

Add this line to enable the optional Ultrasonic Unoccupied To Occupied Delay attribute:

```
#define CLD_OS_ATTR_ULTRASONIC_UNOCCUPIED_TO_OCCUPIED_DELAY
```

Add this line to enable the Ultrasonic PIR Unoccupied To Occupied Threshold attribute:

```
#define CLD_OS_ATTR_ULTRASONIC_UNOCCUPIED_TO_OCCUPIED_THRESHOLD
```

Global Attributes

Add this line to enable the optional Attribute Reporting Status attribute:

```
#define CLD_OS_ATTR_ID_ATTRIBUTE_REPORTING_STATUS
```

Add this line to define the value (n) of the Cluster Revision attribute:

```
#define CLD_OS_CLUSTER_REVISION <n>
```

The default value is 1, which corresponds to the revision of the cluster in the ZCL r6 specification (see Section 2.4).

30 Electrical Measurement Cluster

This chapter outlines the Electrical Measurement cluster, which provides an interface for obtaining electrical measurements from a device.

The Electrical Measurement cluster has a Cluster ID of 0x0B04.

30.1 Overview

The Electrical Measurement cluster provides an interface for querying devices for electrical measurements.

- The server is located on the device which makes the electrical measurements
- The client is located on another device and queries the server for measurements

Separate instances of the Electrical Measurement cluster server can be implemented across multiple endpoints within the same physical unit - that is, one server instance per endpoint. An example is a power extension unit containing multiple outlets, where each power outlet allows electrical measurements to be made on the supplied power

(e.g. AC RMS voltage and current).

The cluster is enabled by defining CLD_ELECTRICAL_MEASUREMENT in the **zcl_options.h** file. Further compile-time options for the Electrical Measurement cluster are detailed in Section 30.9.

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

- Basic Information
- DC Measurement
- DC Formatting
- AC (Non-phase Specific) Measurements
- AC (Non-phase Specific) Formatting
- AC (Single Phase or Phase A) Measurements
- AC Formatting
- DC Manufacturer Threshold Alarms
- AC Manufacturer Threshold Alarms
- AC Phase B Measurements
- AC Phase C Measurements

Note that not all of the above attribute sets are currently implemented in the NXP software and not all attributes within a supported attribute set are implemented (see Section 30.2 for the supported attribute sets and attributes).

30.2 Cluster structure and attributes

The structure definition for the Electrical Measurement cluster (server) is:

```
typedef struct
{
#ifdef ELECTRICAL_MEASUREMENT_SERVER
    zbmap32                 u32MeasurementType;
#ifdef CLD_ELECTMEAS_ATTR_AC_FREQUENCY
    zuint16                 u16ACFrequency;
#endif
#ifdef CLD_ELECTMEAS_ATTR_RMS_VOLTAGE
    zuint16                 u16RMSVoltage;
#endif
```

```
#ifdef CLD_ELECTMEAS_ATTR_RMS_CURRENT
    zuint16                 u16RMSCurrent;
#endif
#ifdef CLD_ELECTMEAS_ATTR_ACTIVE_POWER
    zint16                  i16ActivePower;
#endif
#ifdef CLD_ELECTMEAS_ATTR_REACTIVE_POWER
    zint16                  i16ReactivePower;
#endif
#ifdef CLD_ELECTMEAS_ATTR_APPARENT_POWER
    zuint16                 u16ApparentPower;
#endif
#ifdef CLD_ELECTMEAS_ATTR_POWER_FACTOR
    zint8                   i8PowerFactor;
#endif
#ifdef CLD_ELECTMEAS_ATTR_AC_VOLTAGE_MULTIPLIER
    zuint16                 u16ACVoltageMultiplier;
#endif
#ifdef CLD_ELECTMEAS_ATTR_AC_VOLTAGE_DIVISOR
    zuint16                 u16ACVoltageDivisor;
#endif
#ifdef CLD_ELECTMEAS_ATTR_AC_CURRENT_MULTIPLIER
    zuint16                 u16ACCurrentMultiplier;
#endif
#ifdef CLD_ELECTMEAS_ATTR_AC_CURRENT_DIVISOR
    zuint16                 u16ACCurentDivisor;
#endif
#ifdef CLD_ELECTMEAS_ATTR_AC_POWER_MULTIPLIER
    zuint16                 u16ACPowerMultiplier;
#endif
#ifdef CLD_ELECTMEAS_ATTR_AC_POWER_DIVISOR
    zuint16                 u16ACPowerDivisor;
#endif
#ifdef CLD_ELECTMEAS_ATTR_MAN_SPEC_APPARENT_POWER
    zuint32                  u32ManSpecificApparentPower;
#endif
#ifdef CLD_ELECTMEAS_ATTR_MAN_SPEC_NON_ACTIVE_POWER
    zuint32                  u32NonActivePower;
#endif
#ifdef CLD_ELECTMEAS_ATTR_MAN_SPEC_FNDMTL_REACTIVE_POWER
    zint32                   i32FundamentalReactivePower;
#endif
#ifdef CLD_ELECTMEAS_ATTR_MAN_SPEC_FNDMTL_APPARENT_POWER
    zuint32                  u32FundamentalApparentPower;
#endif
#ifdef CLD_ELECTMEAS_ATTR_MAN_SPEC_FNDMTL_POWER_FACTOR
    zuint16                  u16FundamentalPowerFactor;
#endif
#ifdef CLD_ELECTMEAS_ATTR_MAN_SPEC_NON_FNDMTL_APPARENT_POWER
    zuint32                  u32NonFundamentalApparentPower;
#endif
#ifdef CLD_ELECTMEAS_ATTR_MAN_SPEC_TOTAL_HARMONIC_DISTORTION
    zuint32                  u32TotalHarmonicDistortion;
#endif
#ifdef CLD_ELECTMEAS_ATTR_MAN_SPEC_VBIAS
    zuint32                  u32VBias;
#endif
#ifdef CLD_ELECTMEAS_ATTR_MAN_SPEC_DIVISOR
    zuint16                  u16ManSpecDivisor;
#endif
```

```
#endif
    zuint16                u16ClusterRevision;
}tsCLD_ElectricalMeasurement;
```

where:

‘Basic Information’ Attribute Set

```
u32MeasurementType is a mandatory attribute which is a bitmap indicating the types of elec
```

Bits	Measurement Type
0	Active measurement (AC)
1	Reactive measurement (AC)
2	Apparent measurement (AC)
3	Phase A measurement
4	Phase B measurement
5	Phase C measurement
6	DC measurement
7	Harmonics measurement
8	Power quality measurement
9-31	Reserved

‘AC (Non-phase Specific) Measurements’ Attribute Set

```
u16ACFrequency is an optional attribute containing the most recent measurement of the AC f
```

‘AC (Single Phase or Phase A) Measurements’ Attribute Set

Note that the attributes `u16RMSVoltage`, `u16RMSCurrent` and `i16ActivePower` must be enabled in conjunction with the corresponding multiplier/divisor pair in the ‘AC Formatting’ attribute set.

```
u16RMSVoltage is an optional attribute containing the most recent measurement of the Root
u16RMSCurrent is an optional attribute containing the most recent measurement of the Root
i16ActivePower is an optional attribute containing the present single-
phase or Phase-
A demand for active power, in Watts (W). A positive value represents active power delivere
i16ReactivePower is an optional attribute containing the present single-
phase or Phase-A demand for reactive power, in Volts-Amps-
reactive (VAr). A positive value represents reactive power delivered to the premises and a
u16ApparentPower is an optional attribute containing the present single-
phase or Phase-A demand for apparent power, in Volts-
Amps (VA). This value is the positive square-
root of i16ActivePower squared plus i16ReactivePower squared.
i8PowerFactor is an optional attribute containing the single-phase or
Phase-
A power factor ratio represented as a multiple of 0.01 (e.g. the attribute value 0x0C repr
```

‘AC Formatting’ Attribute Set

The following attributes come in multiplier/divisor pairs, where each pair corresponds to an attribute of the ‘AC (Single Phase or Phase A) Measurements’ attribute set and must only be enabled if the corresponding attribute is enabled.

```
u16ACVoltageMultiplier is an optional attribute containing the multiplication factor to be
u16ACVoltageDivisor is an optional attribute containing the division factor to be applied
u16ACCurrentMultiplier is an optional attribute containing the multiplication factor to be
u16ACCurrentDivisor is an optional attribute containing the division factor to be applied
u16ACPowerMultiplier is an optional attribute containing the multiplication factor to be a
u16ACPowerDivisor is an optional attribute containing the division factor to be applied to
```

Manufacturer-specific Attributes

```
u32ManSpecificApparentPower is an optional manufacturer-
defined attribute containing the demand for apparent power.
u32NonActivePower is an optional manufacturer-
defined attribute containing the demand for non-active power.
i32FundamentalReactivePower is an optional manufacturer-
defined attribute containing the demand for fundamental reactive power.
u32FundamentalApparentPower is an optional manufacturer-
defined attribute containing the demand for fundamental apparent power.
u16FundamentalPowerFactor is an optional manufacturer-
defined attribute representing the power factor of a fundamental power system
u32NonFundamentalApparentPower is an optional manufacturer-
defined attribute representing the power factor of a non-
fundamental (harmonic) power system.
u32TotalHarmonicDistortion is an optional manufacturer-
defined attribute representing the total harmonic distortion present in the delivered powe
u32VBias is an optional manufacturer-
defined attribute representing the bias voltage.
u16ManSpecDivisor is an optional manufacturer-
defined attribute representing a power divisor.
```

Global Attributes

```
u16ClusterRevision is a mandatory attribute that specifies the revision of the cluster spe
```

30.3 Initialisation and Operation

The Electrical Measurement cluster must be initialised on both the cluster server and client. This can be done using the function **eCLD_ElectricalMeasurementCreateElectricalMeasurement()**, which creates an instance of the Electrical Measurement cluster on a local endpoint.

Once the cluster has been initialized, the application on the server should maintain the cluster attributes (see Section 30.2) with the electrical measurements made by the local device. The application on a client can remotely read these measured values using the ZCL ‘Read Attribute’ functions, as described in Section 2.3.2.

30.4 Electrical Measurement Events

There are no events specific to the Electrical Measurement cluster.

30.5 Functions

The following Electrical Measurement cluster function is provided:

30.5.1 eCLD_ElectricalMeasurementCreateElectricalMeasurement

```
teZCL_Status eCLD_ElectricalMeasurementCreateElectricalMeasurement(
        tsZCL_ClusterInstance *psClusterInstance,
        bool_t bIsServer,
        tsZCL_ClusterDefinition *psClusterDefinition,
        void *pvEndPointSharedStructPtr,
        uint8 *pu8AttributeControlBits);
```

Description

This function creates an instance of the Electrical Measurement 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 an Electrical Measurement 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 D.

Note: *This function must not be called for an endpoint on which a standard ZigBee device is 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 the ZigBee Devices User Guide*

Note: *(JNUG3131).*

When used, this function must be called after the stack has been started and after the ZCL has been initialized.

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 is automatically adjusted by the compiler using the following declaration:

```
uint8 au8ElectricalMeasurementAttributeControlBits
[(sizeof(asCLD_ElectricalMeasurementClusterAttributeDefinitions) / sizeof(tsZCL_AttributeD
```

Parameters

psClusterInstance Pointer to structure containing information about the cluster instance to be created (see Section 6.1.16). This structure is updated by the function by initialising individual structure fields.

bIsServer Type of cluster instance (server or client) to be created:

TRUE - server

FALSE - client

psClusterDefinition Pointer to structure indicating the type of cluster to be created (see Section 6.1.2). In this case, this structure must contain the details of the Electrical Measurement cluster. This parameter can refer to a pre-filled structure called `sCLD_ElectricalMeasurement` which is provided in the **ElectricalMeasurement.h** file.

pvEndPointSharedStructPtr Pointer to the shared structure used for attribute storage. This parameter should be the address of the structure of type `tsCLD_ElectricalMeasurement` which defines the attributes of Electrical Measurement cluster. The function initializes the attributes with default values.
pu8AttributeControlBits Pointer to an array of **uint8** values, with one element for each attribute in the cluster (see above). For a cluster client, set this pointer to NULL.

Returns

E_ZCL_SUCCESS
E_ZCL_FAIL
E_ZCL_ERR_PARAMETER_NULL
E_ZCL_ERR_INVALID_VALUE

30.6 Return codes

The Electrical Measurement cluster function uses the ZCL return codes, listed in Section 7.2.

30.7 Enumerations

30.7.1 'Attribute ID' Enumerations

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

```
typedef enum
{
    E_CLD_ELECTMEAS_ATTR_ID_MEASUREMENT_TYPE
    E_CLD_ELECTMEAS_ATTR_ID_AC_FREQUENCY
    E_CLD_ELECTMEAS_ATTR_ID_RMS_VOLTAGE                 = 0x0505,
    E_CLD_ELECTMEAS_ATTR_ID_RMS_CURRENT                 = 0x0508,
    E_CLD_ELECTMEAS_ATTR_ID_ACTIVE_POWER                = 0x050B,
    E_CLD_ELECTMEAS_ATTR_ID_REACTIVE_POWER              = 0x050E,
    E_CLD_ELECTMEAS_ATTR_ID_APPARENT_POWER              = 0x050F,
    E_CLD_ELECTMEAS_ATTR_ID_POWER_FACTOR                = 0x0510,
    E_CLD_ELECTMEAS_ATTR_ID_AC_VOLTAGE_MULTIPLIER       = 0x0600,
    E_CLD_ELECTMEAS_ATTR_ID_AC_VOLTAGE_DIVISOR          = 0x0601,
    E_CLD_ELECTMEAS_ATTR_ID_AC_CURRENT_MULTIPLIER       = 0x0602,
    E_CLD_ELECTMEAS_ATTR_ID_AC_CURRENT_DIVISOR          = 0x0603,
    E_CLD_ELECTMEAS_ATTR_ID_AC_POWER_MULTIPLIER         = 0x0604,
    E_CLD_ELECTMEAS_ATTR_ID_AC_POWER_DIVISOR            = 0x0605,
    E_CLD_ELECTMEAS_ATTR_ID_MAN_SPEC_APPARENT_POWER     = 0xFF00,
    E_CLD_ELECTMEAS_ATTR_ID_MAN_SPEC_NON_ACTIVE_POWER,
    E_CLD_ELECTMEAS_ATTR_ID_MAN_SPEC_FNDMTL_REACTIVE_POWER,
    E_CLD_ELECTMEAS_ATTR_ID_MAN_SPEC_FNDMTL_APPARENT_POWER,
    E_CLD_ELECTMEAS_ATTR_ID_MAN_SPEC_FNDMTL_POWER_FACTOR,
    E_CLD_ELECTMEAS_ATTR_ID_MAN_SPEC_NON_FNDMTL_APPARENT_POWER,
    E_CLD_ELECTMEAS_ATTR_ID_MAN_SPEC_TOTAL_HARMONIC_DISTORTION,
    E_CLD_ELECTMEAS_ATTR_ID_MAN_SPEC_VBIAS,
    E_CLD_ELECTMEAS_ATTR_ID_MAN_SPEC_DIVISOR,
} teCLD_ElectricalMeasurement_AttributeID;
```

30.8 Structures

There are no structures specific to the Electrical Measurement cluster.

30.9 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 Electrical Measurement cluster.

To enable the Electrical Measurement cluster in the code to be built, it is necessary to add the following line to the file:

```
#define CLD_ELECTRICAL_MEASUREMENT
```

In addition, to enable the cluster as a client or server, it is also necessary to add one of the following lines to the same file:

```
#define ELECTRICAL_MEASUREMENT_SERVER
#define ELECTRICAL_MEASUREMENT_CLIENT
```

Optional Attributes

The optional attributes for the Electrical Measurement cluster (see Section 30.2) are enabled by defining:

- CLD_ELECTMEAS_ATTR_AC_FREQUENCY
- CLD_ELECTMEAS_ATTR_RMS_VOLTAGE
- CLD_ELECTMEAS_ATTR_RMS_CURRENT
- CLD_ELECTMEAS_ATTR_ACTIVE_POWER
- CLD_ELECTMEAS_ATTR_REACTIVE_POWER
- CLD_ELECTMEAS_ATTR_APPARENT_POWER
- CLD_ELECTMEAS_ATTR_POWER_FACTOR
- CLD_ELECTMEAS_ATTR_AC_VOLTAGE_MULTIPLIER
- CLD_ELECTMEAS_ATTR_AC_VOLTAGE_DIVISOR
- CLD_ELECTMEAS_ATTR_AC_CURRENT_MULTIPLIER
- CLD_ELECTMEAS_ATTR_AC_CURRENT_DIVISOR
- CLD_ELECTMEAS_ATTR_AC_POWER_MULTIPLIER
- CLD_ELECTMEAS_ATTR_AC_POWER_DIVISOR
- CLD_ELECTMEAS_ATTR_MAN_SPEC_APPARENT_POWER
- CLD_ELECTMEAS_ATTR_MAN_SPEC_NON_ACTIVE_POWER
- CLD_ELECTMEAS_ATTR_MAN_SPEC_FNDMTL_REACTIVE_POWER
- CLD_ELECTMEAS_ATTR_MAN_SPEC_FNDMTL_APPARENT_POWER
- CLD_ELECTMEAS_ATTR_MAN_SPEC_FNDMTL_POWER_FACTOR
- CLD_ELECTMEAS_ATTR_MAN_SPEC_NON_FNDMTL_APPARENT_POWER
- CLD_ELECTMEAS_ATTR_MAN_SPEC_TOTAL_HARMONIC_DISTORTION
- CLD_ELECTMEAS_ATTR_MAN_SPEC_VBIAS
- CLD_ELECTMEAS_ATTR_MAN_SPEC_DIVISOR

Global Attributes

Add this line to define the value (n) of the Cluster Revision attribute:

```
#define CLD_APPLIANCE_STATISTICS_CLUSTER_REVISION <n>
```

The default value is 1, which corresponds to the revision of the cluster in the ZCL r6 specification (see Section 2.4).

Part V: Lighting Clusters

- This part comprises two chapters:
  - Chapter 31 details the **Colour Control** cluster
  - Chapter 32 details the **Ballast Configuration** cluster

31 Colour Control Cluster

This chapter describes the Colour Control cluster which is defined in the ZCL.

The Colour Control cluster has a Cluster ID of 0x0300.

31.1 Overview

The Colour Control cluster is used to control the colour of a light.

Note: ***Note 1:*** *This cluster should normally be used with the Level Control cluster (see Chapter 16) and On/Off cluster (see Chapter 14). This is assumed to be the case in this description.*

Note: ***Note 2:*** *This cluster only controls the colour balance and not the overall brightness of a light. The brightness is adjusted using the Level Control cluster.*

The Colour Control cluster provides the facility to specify the colour of a light in the colour space defined in the *Commission Internationale de l'Éclairage (CIE) specification (1931)*. Colour control can be performed in terms of any of the following:

- x and y values, as defined in the CIE specification
- hue and saturation
- colour temperature

To use the functionality of this cluster, you must include the file **ColourControl.h** in your application and enable the cluster by defining CLD_COLOUR_CONTROL in the **zcl_options.h** file - see Section 31.9.

It is also necessary to enable the cluster as a server or client, or as both:

- The cluster server is able to receive commands to change the colour on the local light device.
- The cluster client is able to send commands to change the colour on the remote light device.

The inclusion of the client or server software must be pre-defined in the application's compile-time options (in addition, if the cluster is to reside on a custom endpoint then the role of client or server must also be specified when creating the cluster instance).

The compile-time options for the Colour Control cluster are fully detailed in Section 31.9.

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

- Colour Information
- Defined Primaries Information
- Additional Defined Primaries Information
- Defined Colour Point Settings
- Enhanced Colour Mode

31.2 Colour Control Cluster structure and attributes

The structure definition for the Colour Control cluster is:

```
typedef struct
{
#ifdef COLOUR_CONTROL_SERVER
    /* Colour information attribute set */
#ifdef CLD_COLOURCONTROL_ATTR_CURRENT_HUE
    zuint8                u8CurrentHue;
#endif
#ifdef CLD_COLOURCONTROL_ATTR_CURRENT_SATURATION
    zuint8                u8CurrentSaturation;
```

```
#endif
#ifdef CLD_COLOURCONTROL_ATTR_REMAINING_TIME
    zuint16               u16RemainingTime;
#endif
    zuint16                 u16CurrentX;
    zuint16                u16CurrentY;
#ifdef CLD_COLOURCONTROL_ATTR_DRIFT_COMPENSATION
    zenum8                u8DriftCompensation;
#endif
#ifdef CLD_COLOURCONTROL_ATTR_COMPENSATION_TEXT
    tsZCL_CharacterString   sCompensationText;
    uint8             au8CompensationText[
        CLD_COLOURCONTROL_COMPENSATION_TEXT_MAX_STRING_LENGTH];
#endif
#ifdef CLD_COLOURCONTROL_ATTR_COLOUR_TEMPERATURE_MIRED
    zuint16               16ColourTemperatureMired;
#endif
#ifdef CLD_COLOURCONTROL_ATTR_COLOUR_MODE
    zenum8                  u8ColourMode;
#endif
    zbmap8                  u8Options;
    /* Defined Primaries Information attribute set */
#ifdef CLD_COLOURCONTROL_ATTR_NUMBER_OF_PRIMARIES
    zuint8               u8NumberOfPrimaries;
#endif
#ifdef CLD_COLOURCONTROL_ATTR_PRIMARY_1_X
    zuint16                 u16Primary1X;
#endif
#ifdef CLD_COLOURCONTROL_ATTR_PRIMARY_1_Y
    zuint16                 u16Primary1Y;
#endif
#ifdef CLD_COLOURCONTROL_ATTR_PRIMARY_1_INTENSITY
    zuint8               u8Primary1Intensity;
#endif
#ifdef CLD_COLOURCONTROL_ATTR_PRIMARY_2_X
    zuint16                 u16Primary2X;
#endif
#ifdef CLD_COLOURCONTROL_ATTR_PRIMARY_2_Y
    zuint16                 u16Primary2Y;
#endif
#ifdef CLD_COLOURCONTROL_ATTR_PRIMARY_2_INTENSITY
    zuint8               u8Primary2Intensity;
#endif
#ifdef CLD_COLOURCONTROL_ATTR_PRIMARY_3_X
    zuint16                 u16Primary3X;
#endif
#ifdef CLD_COLOURCONTROL_ATTR_PRIMARY_3_Y
    zuint16                 u16Primary3Y;
#endif
#ifdef CLD_COLOURCONTROL_ATTR_PRIMARY_3_INTENSITY
    zuint8              u8Primary3Intensity;
#endif
    /* Additional Defined Primaries Information attribute set */
#ifdef CLD_COLOURCONTROL_ATTR_PRIMARY_4_X
    zuint16                 u16Primary4X;
#endif
#ifdef CLD_COLOURCONTROL_ATTR_PRIMARY_4_Y
    zuint16                 u16Primary4Y;
#endif
#ifdef CLD_COLOURCONTROL_ATTR_PRIMARY_4_INTENSITY
```

```
    zuint8                u8Primary4Intensity;
#endif
#ifdef CLD_COLOURCONTROL_ATTR_PRIMARY_5_X
    zuint16                 u16Primary5X;
#endif
#ifdef CLD_COLOURCONTROL_ATTR_PRIMARY_5_Y
    zuint16                 u16Primary5Y;
#endif
#ifdef CLD_COLOURCONTROL_ATTR_PRIMARY_5_INTENSITY
    zuint8                u8Primary5Intensity;
#endif
#ifdef CLD_COLOURCONTROL_ATTR_PRIMARY_6_X
    zuint16                 u16Primary6X;
#endif
#ifdef CLD_COLOURCONTROL_ATTR_PRIMARY_6_Y
    zuint16                 u16Primary6Y;
#endif
#ifdef CLD_COLOURCONTROL_ATTR_PRIMARY_6_INTENSITY
    zuint8                  u8Primary6Intensity;
#endif
    /* Defined Colour Points Settings attribute set */
#ifdef CLD_COLOURCONTROL_ATTR_WHITE_POINT_X
    zuint16               u16WhitePointX;
#endif
#ifdef CLD_COLOURCONTROL_ATTR_WHITE_POINT_Y
    zuint16                 u16WhitePointY;
#endif
#ifdef CLD_COLOURCONTROL_ATTR_COLOUR_POINT_R_X
    zuint16                 u16ColourPointRX;
#endif
#ifdef CLD_COLOURCONTROL_ATTR_COLOUR_POINT_R_Y
    zuint16                 u16ColourPointRY;
#endif
#ifdef CLD_COLOURCONTROL_ATTR_COLOUR_POINT_R_INTENSITY
    zuint8              u8ColourPointRIntensity;
#endif
#ifdef CLD_COLOURCONTROL_ATTR_COLOUR_POINT_G_X
    zuint16             u16ColourPointGX;
#endif
#ifdef CLD_COLOURCONTROL_ATTR_COLOUR_POINT_G_Y
    zuint16                 u16ColourPointGY;
#endif
#ifdef CLD_COLOURCONTROL_ATTR_COLOUR_POINT_G_INTENSITY
    zuint8              u8ColourPointGIntensity;
#endif
#ifdef CLD_COLOURCONTROL_ATTR_COLOUR_POINT_B_X
    zuint16                 u16ColourPointBX;
#endif
#ifdef CLD_COLOURCONTROL_ATTR_COLOUR_POINT_B_Y
    zuint16                 u16ColourPointBY;
#endif
#ifdef CLD_COLOURCONTROL_ATTR_COLOUR_POINT_B_INTENSITY
    zuint8              u8ColourPointBIntensity;
#endif
/* Colour information attribute set */
#ifdef CLD_COLOURCONTROL_ATTR_ENHANCED_CURRENT_HUE
    zuint16                 u16EnhancedCurrentHue;
#endif
#ifdef CLD_COLOURCONTROL_ATTR_ENHANCED_COLOUR_MODE
    zenum8                u8EnhancedColourMode;
```

```
#endif
#ifdef CLD_COLOURCONTROL_ATTR_COLOUR_LOOP_ACTIVE
    zuint8              u8ColourLoopActive;
#endif
#ifdef CLD_COLOURCONTROL_ATTR_COLOUR_LOOP_DIRECTION
    zuint8             u8ColourLoopDirection;
#endif
#ifdef CLD_COLOURCONTROL_ATTR_COLOUR_LOOP_TIME
    zuint16             u16ColourLoopTime;
#endif
#ifdef CLD_COLOURCONTROL_ATTR_COLOUR_LOOP_START_ENHANCED_HUE
    zuint16             u16ColourLoopStartEnhancedHue;
#endif
#ifdef CLD_COLOURCONTROL_ATTR_COLOUR_LOOP_STORED_ENHANCED_HUE
    zuint16                 u16ColourLoopStoredEnhancedHue;
#endif
#ifdef CLD_COLOURCONTROL_ATTR_COLOUR_CAPABILITIES
    zuint16                 u16ColourCapabilities;
#endif
#ifdef CLD_COLOURCONTROL_ATTR_COLOUR_TEMPERATURE_MIRED_PHY_MIN
    zuint16           u16ColourTemperatureMiredPhyMin;
#endif
#ifdef CLD_COLOURCONTROL_ATTR_COLOUR_TEMPERATURE_MIRED_PHY_MAX
    zuint16          u16ColourTemperatureMiredPhyMax;
#endif
#ifdef CLD_COLOURCONTROL_ATTR_COUPLE_COLOUR_TEMPERATURE_TO_LEVEL_MIN_MIRED
    zuint16    u16CoupleColourTempToLevelMinMired;
#endif
#ifdef CLD_COLOURCONTROL_ATTR_STARTUP_COLOUR_TEMPERATURE_MIRED
    zuint16          u16StartupColourTemperatureMired;
#endif
#ifdef CLD_COLOURCONTROL_ATTR_ATTRIBUTE_REPORTING_STATUS
    zenum8           u8AttributeReportingStatus;
#endif
#endif
    zuint16           u16ClusterRevision;
} tsCLD_ColourControl;
```

where:

‘Colour Information’ Attribute Set

Note that the attributes `u8CurrentHue`, `u8CurrentSaturation`, `u16CurrentX`, `u16CurrentY` and `u16ColourTemperatureMired` are enabled as part of ‘Colour Capabilities’ groups - see Table 34 on page 704.

- `u8CurrentHue` is the current hue value of the light in the range 0-254. This value can be converted to hue in degrees using the following formula:
  hue = `u8CurrentHue` x 360/254. This attribute is only valid when the attributes `u8CurrentSaturation` and `u8ColorMode` are also implemented.
- `u8CurrentSaturation` is the current saturation value of the light in the range 0-254. This value can be converted to saturation as a fraction using the following formula: saturation = `u8CurrentSaturation`/254. This attribute is only valid when the attributes `u8CurrentHue` and `u8ColorMode` are also implemented.
- `u16RemainingTime` is the time duration, in tenths of a second, before the currently active command completes.
- `u16CurrentX` is the current value for the chromaticity x, as defined in the CIE xyY colour space, in the range 0-65279. The normal value of x is calculated using the following formula: x = `u16CurrentX`/65536.

- u16CurrentY is the current value for the chromaticity y, as defined in the CIE xyY colour space, in the range 0-65279. The normal value of y is calculated using the following formula: y = u16CurrentY/65536.
- u8DriftCompensation indicates the mechanism, if any, is being used to compensate for colour/intensity drift over time. One of the following values is specified:

Table 43. u8DriftCompensation attribute bit values

u8DriftCompensation	Drift Compensation Mechanism
0x00	None
0x01	Other or unknown
0x02	Temperature monitoring
0x03	Optical luminance monitoring and feedback
0x04	Optical colour monitoring and feedback
0x05 - 0xFF	Reserved

- The following optional pair of attributes are used to store a textual indication of the drift compensation mechanism used:
  - sCompensationText is a tsZCL_CharacterString structure (see Section 6.1.14) for a character string representing the drift compensation method used
  - au8CompensationText[] is a byte-array which contains the character data bytes representing the drift compensation method used
- u16ColourTemperatureMired is the colour temperature of the light expressed as a micro reciprocal degree (mired) value. It is a scaled reciprocal of the current value of the colour temperature, in the range 1-65279
  (0 is undefined and 65535 indicates an invalid value). The colour temperature, in Kelvin, is calculated using the following formula:
  T = 1000000/u16ColourTemperatureMired. This attribute is only valid when the attribute u8ColourMode is also implemented.
- u8ColourMode indicates which method is currently being used to control the colour of the light. One of the following values is specified:

Table 44. u8ColourMode attribute bit values

u8ColourMode	Colour Control Method/Attributes
0x00	Hue and saturation (u8CurrentHue and u8CurrentSaturation)
0x01	Chromaticities x and y from CIE xyY colour space (u16CurrentX and u16CurrentY)
0x02	Colour temperature (u16ColourTemperatureMired)
0x03 - 0xFF	Reserved

- u8Options is a bitmap which allows behaviors connected with certain commands to be defined (these behaviors should only be defined during commissioning), as follows:

Bits	Name	Description
0	ChangeIfOff	Defines whether changes to the Colour Control cluster can be made from control clusters (e.g. Level Control) when the bOnOff attribute of the On/Off cluster is zero (off): • 1 – Allow changes • 0 – Do not allow changes

Bits	Name	Description
1-7	-	Reserved

‘Defined Primaries Information’ Attribute Set

- `u8NumberOfPrimaries` is the number of colour primaries implemented on the device, in the range 1-6 (0xFF is used if the number of primaries is unknown).
  - For each colour primary, there is a set of three attributes used (see below) - for example, for the first primary this attribute trio comprises `u16Primary1X`, `u16Primary1Y` and `u8Primary1Intensity`. Therefore, the number of primaries specified determines the number of these attribute trios used.

Note: *The number of primaries is set using a macro at compile-time (see Section 31.9). This automatically enables the relevant `u16PrimaryNX`, `u16PrimaryNY` and `u8PrimaryNIntensity` (N=1 to 6) attributes.*

- The attribute definitions below are valid for colour primary N, where N is 1, 2 or 3.
- `u16PrimaryNX` is the value for the chromaticity x for colour primary N, as defined in the CIE xyY colour space, in the range 0-65279. The normalized value of x is calculated using the following formula: x = `u16PrimaryNX`/65536.
- `u16PrimaryNY` is the value for the chromaticity y for colour primary N, as defined in the CIE xyY colour space, in the range 0-65279. The normalized value of y is calculated using the following formula: y = `u16PrimaryNY`/65536.
- `u8PrimaryNIntensity` is a representation of the maximum intensity of colour primary 1, normalized such that the primary with the highest maximum intensity has the value 0xFE.

‘Additional Defined Primaries Information’ Attribute Set

- The attribute definitions for this set are as for `u16PrimaryNX`, `u16PrimaryNY` and `u8PrimaryNIntensity` above, where N is 4, 5 or 6.
- As indicated in the Note above for the ‘Defined Primaries Information’ Attribute Set, these attributes are enabled automatically according to the number of required primaries defined at compile-time (see Section 31.9).

‘Defined Colour Points Settings’ Attribute Set

- `u16WhitePointX` is the value for the chromaticity x for the white point of the device, as defined in the CIE xyY colour space, in the range 0-65279. The normalized value of x is calculated using the following formula: x = `u16WhitePointX`/65536.
- `u16WhitePointY` is the value for the chromaticity y for the white point of the device, as defined in the CIE xyY colour space, in the range 0-65279. The normalized value of y is calculated using the following formula: y = `u16WhitePointY`/65536.
- `u16ColourPointRX` is the value for the chromaticity x for the red colour point of the device, as defined in the CIE xyY colour space, in the range 0-65279. The normalized value of x is calculated using the following formula:
  x = `u16ColourPointRX`/65536.
- `u16ColourPointRY` is the value for the chromaticity y for the red colour point of the device, as defined in the CIE xyY colour space, in the range 0-65279. The normalized value of y is calculated using the following formula:
  y = `u16ColourPointRY`/65536.
- `u8ColourPointRIntensity` is a representation of the relative intensity of the red colour point of the device, normalized such that the colour point with the highest relative intensity has the value 0xFE (the value 0xFF indicates an invalid value).

- u16ColourPointGX is the value for the chromaticity x for the green colour point of the device, as defined in the CIE xyY colour space, in the range 0-65279. The normalized value of x is calculated using the following formula:
  x = u16ColourPointGX/65536.
- u16ColourPointGY is the value for the chromaticity y for the green colour point of the device, as defined in the CIE xyY colour space, in the range 0-65279. The normalized value of y is calculated using the following formula:
  y = u16ColourPointGY/65536.
- u8ColourPointGIntensity is a representation of the relative intensity of the green colour point of the device, normalized such that the colour point with the highest relative intensity has the value 0xFE (the value 0xFF indicates an invalid value).
- u16ColourPointBX is the value for the chromaticity x for the blue colour point of the device, as defined in the CIE xyY colour space, in the range 0-65279. The normalized value of x is calculated using the following formula:
  x = u16ColourPointBX/65536.
- u16ColourPointBY is the value for the chromaticity y for the blue colour point of the device, as defined in the CIE xyY colour space, in the range 0-65279. The normalized value of y is calculated using the following formula:
  y = u16ColourPointBY/65536.
- u8ColourPointBIntensity is a representation of the relative intensity of the blue colour point of the device, normalized such that the colour point with the highest relative intensity has the value 0xFE (the value 0xFF indicates an invalid value).

Enhanced Colour Mode Attributes

These attributes are enabled as part of ‘Colour Capabilities’ groups - see Table 34 on page 704.

- u16EnhancedCurrentHue contains the current hue of the light in terms of (unequal) steps around the CIE colour ‘triangle’:
  - 8 most significant bits represent an index into the XY look-up table that contains the step values, thus indicating the current step used
  - 8 least significant bits represent a linear interpolation value between the current step and next step (up), facilitating a colour zoom
    - The value of the u8CurrentHue attribute is calculated from the above values.
- u8EnhancedColourMode indicates which method is currently being used to control the colour of the light. One of the following values is specified:

Table 45. u8EnhancedColourMode attribute

u8ColourMode	Colour Control Method/Attributes
0x00	Current hue and current saturation (u8CurrentHue and u8CurrentSaturation)
0x01	Chromaticities x and y from CIE xyY colour space (u16CurrentX and u16CurrentY)
0x02	Colour temperature (u16ColourTemperatureMired)
0x03	Enhanced hue and current saturation (u16EnhancedCurrentHue and u8CurrentSaturation)
0x03 - 0xFF	Reserved

- u8ColourLoopActive indicates whether the colour loop is currently active: 0x01 - active, 0x00 - not active (all other values are reserved). The colour loop follows the hue steps around the CIE colour ‘triangle’ by incrementing or decrementing the value of u16EnhancedCurrentHue.
- u8ColourLoopDirection indicates the current direction of the colour loop in terms of the direction of change of u16EnhancedCurrentHue:
  0x01 - incrementing, 0x00 - decrementing (all other values are reserved).
- u16ColourLoopTime is the period, in seconds, of a full colour loop - that is, the time to cycle all possible values of u16EnhancedCurrentHue.
- u16ColourLoopStartEnhancedHue indicates the value of u16EnhancedCurrentHue at which the colour loop must be started.
- u16ColourLoopStoredEnhancedHue contains the value of u16EnhancedCurrentHue at which the last colour loop completed (this value is stored on completing a colour loop).
- u16ColourCapabilities is a bitmap indicating the Colour Control cluster features (and attributes) supported by the device, as detailed below (a bit is set to ‘1’ if the feature is supported or ‘0’ otherwise):

Table 46.

Bits	Feature	Attributes
0	Hue/Saturation	u8CurrentHue u8CurrentSaturation
1	Enhanced Hue (Hue/Saturation must also be supported)	u16EnhancedCurrentHue
2	Colour Loop (Enhanced Hue must also be supported)	u8ColourLoopActive u8ColourLoopDirection u16ColourLoopTime u16ColourLoopStartEnhancedHue u16ColourLoopStoredEnhancedHue u16ColourCapabilities
3	CIE XY Values	u16CurrentX u16CurrentY
4	Colour Temperature (Mired)	u16ColourTemperatureMired u16ColourTemperatureMiredPhyMin u16ColourTemperatureMiredPhyMax
5-15	Reserved	-

Macros are provided to select the required Colour Capabilities at compile-time - see Table 34 on page 704.

- u16ColourTemperatureMiredPhyMin indicates the minimum value (supported by the hardware) of the mired colour temperature attribute.
- u16ColourTemperatureMiredPhyMax indicates the maximum value (supported by the hardware) of the mired colour temperature attribute.
- u16CoupleColourTempToLevelMinMired is an optional attribute that is used when the u16ColourTemperatureMired attribute is coupled to the u8CurrentLevel attribute of the Level Control cluster (this is the case when the CoupleColorTempToLevel bit of the u8Options attribute of the Level Control cluster is equal to 1). u16CoupleColourTempToLevelMinMired specifies a lower bound on the value of the u16ColourTemperatureMired attribute, where this lower bound corresponds to a u8CurrentLevel value of 0xFE (100%). Note that because the colour temperature is represented as a mired (reciprocal) value, a high value of u8CurrentLevel corresponds to a low value of

`u16ColourTemperatureMired` and the `16CoupleColourTempToLevelMinMired` attribute corresponds to an upper bound on the value of the colour temperature supported by the device. The value of this attribute must be at least equal to the value of `u16ColourTemperatureMiredPhyMin`.

- `u16StartupColourTemperatureMired` is an optional attribute to define the required start-up colour temperature of a light when it is supplied with power. It determines the initial value of `u16ColourTemperatureMired` on start-up.

Global Attributes

- `u8AttributeReportingStatus` is an optional attribute that should be enabled when attribute reporting is used for the cluster (see Section 2.3.5). The value of this attribute indicates whether there are attribute reports still pending (0x00) or the attribute reports are complete (0x01) - all other values are reserved. This attribute is also described in Section 2.4.

u16ClusterRevision is a mandatory attribute that specifies the revision of the cluster specification on which this cluster instance is based. The cluster specification in the ZCLr6 corresponds to a cluster revision of 1. The value is incremented by one for each subsequent revision of the cluster specification. This attribute is also described inSection2.4.

31.3 Attributes for Default Reporting

The following attributes of the Colour Control cluster can be selected for default reporting:

```
u8CurrentHue
u8CurrentSaturation
u16CurrentX
```

- `u16CurrentY`

Attribute reporting (including default reporting) is described in Appendix B. Enabling reports for these attributes is described in Appendix B.3.6.

31.4 Initialization

The function **eCLD_ColourControlCreateColourControl()** is used to create an instance of the Colour Control cluster. The function is generally called by the initialization function for the host device.

31.5 Sending Commands

The NXP implementation of the ZCL provides functions for sending commands between a Colour Control cluster client and server. A command is sent from the client to one or more endpoints on the server. Multiple endpoints can usually be targeted using binding or group addressing.

Note: *Any 'Move to', 'Move' or 'Step' command that is currently in progress can be stopped at any time by calling the function:* ***eCLD_ColourControlCommandStopMoveStepCommandSend()***

31.5.1 Controlling Hue

Colour can be controlled in terms of hue, which is related to the dominant wavelength (or frequency) of the light emitted by a lighting device. On a device that supports the Colour Control cluster, the hue is controlled by means of the 'current hue' attribute (`u8CurrentHue`) of the cluster. This attribute can take a value in the range 0-254, which can be converted to hue in degrees using the following formula:

Hue in degrees = `u8CurrentHue` x 360/254

The ‘current hue’ attribute can be controlled in a number of ways using commands of the Colour Control cluster. API functions are available to send these commands to endpoints on remote devices.

‘Move to Hue’ Command

The ‘Move to Hue’ command allows the ‘current hue’ attribute to be moved (increased or decreased) to a specified target value in a continuous manner over a specified transition time. This command can be sent to an endpoint on a remote device using the function

eCLD_ColourControlCommandMoveToHueCommandSend()

Since the possible hues are represented on a closed boundary, the target hue can be reached by moving the attribute value in either direction, up or down (the attribute value wraps around). Options are also provided for taking the ‘shortest route’ and ‘longest route’ around the boundary.

‘Move Hue’ Command

The ‘Move Hue’ command allows the ‘current hue’ attribute to be moved in a given direction (increased or decreased) at a specified rate indefinitely, until stopped. This command can be sent to an endpoint on a remote device using the function

eCLD_ColourControlCommandMoveHueCommandSend()

Since the possible hues are represented on a closed boundary, the movement is cyclic (the attribute value wraps around). The above function can also be used to stop the movement.

‘Step Hue’ Command

The ‘Step Hue’ command allows the ‘current hue’ attribute to be moved (increased or decreased) by a specified amount in a continuous manner over a specified transition time. This command can be sent to an endpoint on a remote device using the function

eCLD_ColourControlCommandStepHueCommandSend()

Note: *Hue can also be moved in conjunction with saturation, as described in Section 31.5.7. The ‘enhanced’ hue can be moved in similar ways, as described in Section 31.5.5.*

31.5.2 Controlling Saturation

Colour can be controlled in terms of saturation, which is related to the spread of wavelengths (or frequencies) in the light emitted by a lighting device. On a device that supports the Colour Control cluster, the saturation is controlled by means of the ‘current saturation’ attribute (`u8CurrentSaturation`) of the cluster. This attribute can take a value in the range 0-254, which can be converted to saturation as a fraction using the following formula:

Saturation = `u8CurrentSaturation`/254

The ‘current saturation’ attribute can be controlled in a number of ways using commands of the Colour Control cluster. API functions are available to send these commands to endpoints on remote devices.

‘Move to Saturation’ Command

The ‘Move to Saturation’ command allows the ‘current saturation’ attribute to be moved (increased or decreased) to a specified target value in a continuous manner over a specified transition time. This command can be sent to an endpoint on a remote device using the function

eCLD_ColourControlCommandMoveToSaturationCommandSend()

‘Move Saturation’ Command

The ‘Move Saturation’ command allows the ‘current saturation’ attribute to be moved in a given direction (increased or decreased) at a specified rate until stopped or until the current saturation reaches its minimum or maximum value. This command can be sent to an endpoint on a remote device using the function

eCLD_ColourControlCommandMoveSaturationCommandSend()

The above function can also be used to stop the movement.

‘Step Saturation’ Command

The ‘Step Saturation’ command allows the ‘current saturation’ attribute to be moved (increased or decreased) by a specified amount in a continuous manner over a specified transition time. This command can be sent to an endpoint on a remote device using the function

eCLD_ColourControlCommandStepSaturationCommandSend()

Note: *Saturation can also be moved in conjunction with hue, as described in Section 31.5.7.*

31.5.3 Controlling Colour (CIE x and y Chromaticities)

Colour can be controlled in terms of the x and y chromaticities defined in the CIE xyY colour space. On a device that supports the Colour Control cluster, these values are controlled by means of the ‘current x’ attribute (`u16CurrentX`) and ‘current y’ attribute (`u16CurrentY`) of the cluster. Each of these attributes can take a value in the range 0-65279. The normalized x and y chromaticities can then be calculated from these values using the following formulae:

x = `u16CurrentX`/65536

y = `u16CurrentY`/65536

The x and y chromaticity attributes can be controlled in a number of ways using commands of the Colour Control cluster. API functions are available to send these commands to endpoints on remote devices.

‘Move to Colour’ Command

The ‘Move to Colour’ command allows the ‘current x’ and ‘current y’ attributes to be moved (increased or decreased) to specified target values in a continuous manner over a specified transition time. This command can be sent to an endpoint on a remote device using the function

eCLD_ColourControlCommandMoveToColourCommandSend()

‘Move Colour’ Command

The ‘Move Colour’ command allows the ‘current x’ and ‘current y’ attributes to be moved in a given direction (increased or decreased) at specified rates until stopped or until both attributes reach their minimum or maximum value. This command can be sent to an endpoint on a remote device using the function

eCLD_ColourControlCommandMoveColourCommandSend()

The above function can also be used to stop the movement.

‘Step Colour’ Command

The ‘Step Colour’ command allows the ‘current x’ and ‘current y’ attributes to be moved (increased or decreased) by specified amounts in a continuous manner over a specified transition time. This command can be sent to an endpoint on a remote device using the function

eCLD_ColourControlCommandStepColourCommandSend()

31.5.4 Controlling Colour Temperature

Colour can be controlled in terms of colour temperature, which is the temperature of an ideal black body which radiates light of a similar hue to that of the lighting device. On a device that supports the Colour Control cluster, the colour temperature is controlled by means of the 'mired colour temperature' attribute (`u16ColourTemperatureMired`) of the cluster. This attribute stores a micro reciprocal degree (mired) value, which is a scaled reciprocal of the current value of the colour temperature of the light, in the range 1-65279. The colour temperature, in Kelvin, can be calculated from the attribute value using the following formula:

T = 1000000/`u16ColourTemperatureMired`

Note: *The movement of colour temperature through colour space always follows the 'Black Body Line'.*

'Move to Colour Temperature' Command

The 'Move to Colour Temperature' command allows the 'mired colour temperature' attribute to be moved (increased or decreased) to a specified target value in a continuous manner over a specified transition time. This command can be sent to an endpoint on a remote device using the function

eCLD_ColourControlCommandMoveToColourTemperatureCommandSend()

'Move Colour Temperature' Command

The 'Move Colour Temperature' command allows the 'mired colour temperature' attribute to be moved in a given direction (increased or decreased) at a specified rate until stopped. This command can be sent to an endpoint on a remote device using the function

eCLD_ColourControlCommandMoveColourTemperatureCommandSend()

The above function can also be used to stop the movement.

Maximum and minimum values for the 'mired colour temperature' attribute during the movement are also specified. If the attribute value reaches the specified maximum or minimum before the required change has been achieved, the movement will automatically stop.

'Step Colour Temperature' Command

The 'Step Colour Temperature' command allows the 'mired colour temperature' attribute to be moved (increased or decreased) by a specified amount in a continuous manner over a specified transition time. This command can be sent to an endpoint on a remote device using the function

eCLD_ColourControlCommandStepColourTemperatureCommandSend()

Maximum and minimum values for the 'mired colour temperature' attribute during the movement are also specified. If the attribute value reaches the specified maximum or minimum before the required change has been achieved, the movement will automatically stop.

31.5.5 Controlling 'Enhanced' Hue

Colour can be controlled in terms of hue, which is related to the dominant wavelength (or frequency) of the light emitted by a lighting device. The hue can alternatively be controlled by means of the 'enhanced current hue' attribute (`u16EnhancedCurrentHue`), instead of the 'current hue' attribute (the 'current hue' attribute is automatically adjusted when the 'enhanced current hue' attribute value changes).

The ‘enhanced current hue’ attribute allows hue to be controlled on a finer scale than the ‘current hue’ attribute. Hue steps are defined in a look-up table and values between the steps can be achieved through linear interpolation. This 16-bit attribute value therefore comprises two 8-bit components, as described below.

Table 47. ‘Enhanced Current Hue’ Attribute Format

Bits 15-8	Bits 7-0
Index into the look-up table that contains the hue step values, thus indicating the current step used	Linear interpolation value between the current step and next step (up)

Thus, if the current hue step value is H_i (where i is the relevant table index) and the linear interpolation value is *interp*, the ‘enhanced’ hue is given by the formula:

Enhanced hue = H_i + (*interp*/255) x (H_{i+1} - H_i)

To convert this hue to a value in degrees, it is then necessary to multiply by 360/255.

The ‘enhanced current hue’ attribute can be controlled in a number of ways using commands of the Colour Control cluster. API functions are available to send these commands to endpoints on remote devices.

Note: ***Note:*** *These commands are issued by a cluster client and are performed on a cluster server. The look-up table is user-defined on the server. When this command is received by the server, the user-defined callback function that is invoked must read the entry with the specified index from the look-up table and calculate the corresponding ‘enhanced’ hue value.*

‘Enhanced Move to Hue’ Command

The ‘Enhanced Move to Hue’ command allows the ‘enhanced current hue’ attribute to be moved (increased or decreased) to a specified target value in a continuous manner over a specified transition time (the ‘current hue’ attribute is also moved to a value based on the target ‘enhanced current hue’ value). This command can be sent to an endpoint on a remote device using the function

eCLD_ColourControlCommandEnhancedMoveToHueCommandSend()

Since the possible hues are represented on a closed boundary, the target hue can be reached by moving the attribute value in either direction, up or down (the attribute value wraps around). Options are also provided for taking the ‘shortest route’ and ‘longest route’ around the boundary.

‘Enhanced Move Hue’ Command

The ‘Enhanced Move Hue’ command allows the ‘enhanced current hue’ attribute to be moved in a given direction (increased or decreased) at a specified rate indefinitely, until stopped (the ‘current hue’ attribute is also moved through values based on the ‘enhanced current hue’ value). This command can be sent to an endpoint on a remote device using the function

eCLD_ColourControlCommandEnhancedMoveHueCommandSend()

The above function can also be used to stop the movement.

Since the possible hues are represented on a closed boundary, the movement is cyclic (the attribute value wraps around). The above function can also be used to stop the movement.

‘Enhanced Step Hue’ Command

The ‘Enhanced Step Hue’ command allows the ‘enhanced current hue’ attribute to be moved (increased or decreased) by a specified amount in a continuous manner over a specified transition time (the ‘current hue’

attribute is also moved through values based on the ‘enhanced current hue’ value). This command can be sent to an endpoint on a remote device using the function

eCLD_ColourControlCommandEnhancedStepHueCommandSend()

Note: ***Note 1:*** *‘Enhanced’ hue can also be moved in conjunction with saturation, as described in Section 31.5.7.*

Note: ***Note 2:*** *The value of the ‘enhanced current hue’ attribute can be moved around a colour loop, as described in Section 31.5.6.*

31.5.6 Controlling a Colour Loop

The colour of a device can be controlled by moving the value of the ‘enhanced current hue’ attribute around a colour loop corresponding to the CIE colour ‘triangle’ - refer to Section 31.5.5 for details of the ‘enhanced current hue’ attribute.

Movement along the colour loop can be controlled using the ‘Colour Loop Set’ command of the Colour Control cluster. A function is available to send this command to endpoints on remote devices.

‘Colour Loop Set’ Command

The ‘Colour Loop Set’ command allows movement of the ‘enhanced current hue’ attribute value around the colour loop to be configured and started. The direction(up or down), start ‘enhanced’ hue and duration of the movement can be specified. This command can be sent to an endpoint on a remote device using the function

eCLD_ColourControlCommandColourLoopSetCommandSend()

The above function can also be used to stop the movement.

31.5.7 Controlling Hue and Saturation

Colour can be completely specified in terms of hue and saturation, which respectively represent the dominant wavelength (or frequency) of the light and the spread of wavelengths (around the former) within the light. Therefore, the Colour Control cluster provides commands to change both the hue and saturation at the same time. In fact, commands are provided to control the values of the:

- ‘current hue’ and ‘current saturation’ attributes
- ‘enhanced current hue’ and ‘current saturation’ attributes

API functions are available to send these commands to endpoints on remote devices.

‘Move to Hue and Saturation’ Command

The ‘Move to Hue and Saturation’ command allows the ‘current hue’ and ‘current saturation’attributes to be moved to specified target values in a continuous manner over a specified transition time. This command can be sent to an endpoint on a remote device using the function

eCLD_ColourControlCommandMoveToHueCommandSend()

‘Enhanced Move to Hue and Saturation’ Command

The ‘Enhanced Move to Hue and Saturation’ command allows the ‘enhanced current hue’ and ‘current saturation’attributes to be moved to specified target values in a continuous manner over a specified transition time. This command can be sent to an endpoint on a remote device using the function

eCLD_ColourControlCommandEnhancedMoveToHueAndSaturationCommandSend()

31.6 Functions

The following Colour Control cluster functions are provided in the NXP implementation of the ZCL:

- eCLD_ColourControlCreateColourControl
- eCLD_ColourControlCommandMoveToHueCommandSend
- eCLD_ColourControlCommandMoveHueCommandSend
- eCLD_ColourControlCommandStepHueCommandSend
- eCLD_ColourControlCommandMoveToSaturationCommandSend
- eCLD_ColourControlCommandMoveSaturationCommandSend
- eCLD_ColourControlCommandStepSaturationCommandSend
- eCLD_ColourControlCommandMoveToHueAndSaturationCommandSend
- eCLD_ColourControlCommandMoveToColourCommandSend
- eCLD_ColourControlCommandMoveColourCommandSend
- eCLD_ColourControlCommandStepColourCommandSend
- eCLD_ColourControlCommandEnhancedMoveToHueCommandSend
- eCLD_ColourControlCommandEnhancedMoveHueCommandSend
- eCLD_ColourControlCommandEnhancedStepHueCommandSend
- eCLD_ColourControlCommandEnhancedMoveToHueAndSaturationCommandSend
- eCLD_ColourControlCommandColourLoopSetCommandSend
- eCLD_ColourControlCommandStopMoveStepCommandSend
- eCLD_ColourControlCommandMoveToColourTemperatureCommandSend
- eCLD_ColourControlCommandMoveColourTemperatureCommandSend
- eCLD_ColourControlCommandStepColourTemperatureCommandSend
- eCLD_ColourControl_GetRGB

31.6.1 eCLD_ColourControlCreateColourControl

```
teZCL_Status eCLD_ColourControlCreateColourControl(
        tsZCL_ClusterInstance *psClusterInstance,
        bool_t bIsServer,
        tsZCL_ClusterDefinition *psClusterDefinition,
        void *pvEndPointSharedStructPtr,
        uint8 *pu8AttributeControlBits,
        tsCLD_ColourControlCustomDataStructure
         *psCustomDataStructure);
```

Description

This function creates an instance of the Colour Control 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 creates a Colour Control cluster instance on the endpoint, but instances of other clusters may also be created on the same endpoint by calling their corresponding creation functions.

Note: *This function must not be called for an endpoint on which a standard ZigBee device is used. In this case, the device and its supported clusters must be registered on the endpoint using the relevant device registration function.*

When used, this function must be the first Colour Control cluster function called in the application, and must be called after the stack has been started and after the ZCL has been initialized.

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 Colour Control cluster. The function initializes the array elements to zero.

Parameters

- *psClusterInstance*: Pointer to structure containing information about the cluster instance to be created (see Section 6.1.16). This structure is updated by the function by initializing individual structure fields.
- *bIsServer* : Type of cluster instance (server or client) to be created:
- TRUE - server
- FALSE - client
- *psClusterDefinition*: Pointer to structure indicating the type of cluster to be created (see Section 6.1.2). In this case, this structure must contain the details of the Colour Control cluster. This parameter can refer to a pre-filled structure called `sCLD_ColourControl` which is provided in the **ColourControl.h** file.
- *pvEndPointSharedStructPtr*: Pointer to the shared structure used for attribute storage. This parameter should be the address of the structure of type `tsCLD_ColourControl` which defines the attributes of Colour Control cluster. The function initializes the attributes with default values.
- *pu8AttributeControlBits*: Pointer to an array of **uint8** values, with one element for each attribute in the cluster (see above).
- *psCustomDataStructure*: Pointer to a structure containing the storage for internal functions of the cluster (see Section 31.7.1)

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

31.6.2 eCLD_ColourControlCommandMoveToHueCommandSend

```
teZCL_Status eCLD_ColourControlCommandMoveToHueCommandSend(
        uint8 u8SourceEndPointId,
        uint8 u8DestinationEndPointId,
        tsZCL_Address *psDestinationAddress,
        uint8 *pu8TransactionSequenceNumber,
        tsCLD_ColourControl_MoveToHueCommandPayload
 *psPayload);
```

Description

This function sends a Move to Hue command to instruct a device to move its ‘current hue’ attribute to a target hue value in a continuous manner within a given time. The hue value, direction and transition time are specified in the payload of the command (see Section 31.7.2).

Since the possible hues are represented on a closed boundary, the target hue can be reached by moving the attribute value in either direction, up or down (the attribute value wraps around). Options are also provided for ‘shortest route’ and ‘longest route’ around the boundary.

The device receiving this message generates a callback event on the endpoint on which the Colour Control cluster was registered. The device must first ensure that ‘hue and saturation’ mode is selected by setting the ‘colour mode’ attribute to 0x00, if required. It can then move the ‘current hue’ value as requested.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is 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.

This function can only be used when the ‘current hue’ attribute is enabled in the Colour Control cluster.

Parameters

- *u8SourceEndPointId*: Number of the local endpoint through which to send the request. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values
- *u8DestinationEndPointId*: Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP
- *psDestinationAddress*: Pointer to a structure holding the address of the node to which the request is sent
- *pu8TransactionSequenceNumber*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the request
- *psPayload*: Pointer to a structure containing the payload for this message (see Section 31.7.2)

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

This function invokes the ZigBee PRO stack function to transmit the data. In case an error is returned, call the **eZCL_GetLastZpsError()** function to get the error.

31.6.3 eCLD_ColourControlCommandMoveHueCommandSend

```
teZCL_Status eCLD_ColourControlCommandMoveHueCommandSend(
        uint8 u8SourceEndPointId,
        uint8 u8DestinationEndPointId,
        tsZCL_Address *psDestinationAddress,
        uint8 *pu8TransactionSequenceNumber,
        tsCLD_ColourControl_MoveHueCommandPayload
 *psPayload);
```

Description

This function sends a Move Hue command to instruct a device to move its 'current hue' attribute value in a given direction at a specified rate for an indefinite time. The direction and rate are specified in the payload of the command (see Section 31.7.2).

The command can request that the hue is moved up or down, or that existing movement is stopped. Since the possible hues are represented on a closed boundary, the movement is cyclic (the attribute value wraps around). Once started, the movement will continue until it is stopped.

The device receiving this message will generate a callback event on the endpoint on which the Colour Control cluster was registered. The device must first ensure that 'hue and saturation' mode is selected by setting the 'colour mode' attribute to 0x00, if required. It can then move the 'current hue' value as requested.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is 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.

This function can only be used when the 'current hue' attribute is enabled in the Colour Control cluster.

Parameters

- *u8SourceEndPointId*: Number of the local endpoint through which to send the request. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values
- *u8DestinationEndPointId*: Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP
- *psDestinationAddress*: Pointer to a structure holding the address of the node to which the request is sent
- *pu8TransactionSequenceNumber*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the request
- *psPayload*: Pointer to a structure containing the payload for this message (see Section 31.7.2)

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

31.6.4 eCLD_ColourControlCommandStepHueCommandSend

```
teZCL_Status eCLD_ColourControlCommandStepHueCommandSend(
        uint8 u8SourceEndPointId,
        uint8 u8DestinationEndPointId,
        tsZCL_Address *psDestinationAddress,
        uint8 *pu8TransactionSequenceNumber,
        tsCLD_ColourControl_StepHueCommandPayload
 *psPayload);
```

Description

This function sends a Step Hue command to instruct a device to increase or decrease its ‘current hue’ attribute by a specified ‘step’ value in a continuous manner within a given time. The step size, direction and transition time are specified in the payload of the command (see Section 31.7.2).

The device receiving this message will generate a callback event on the endpoint on which the Colour Control cluster was registered. The device must first ensure that ‘hue and saturation’ mode is selected by setting the ‘colour mode’ attribute to 0x00, if required. It can then move the ‘current hue’ value as requested.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is 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.

This function can only be used when the ‘current hue’ attribute is enabled in the Colour Control cluster.

Parameters

- *u8SourceEndPointId*: Number of the local endpoint through which to send the request. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values
- *u8DestinationEndPointId*: Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP
- *psDestinationAddress*: Pointer to a structure holding the address of the node to which the request is sent
- *pu8TransactionSequenceNumber*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the request
- *psPayload*: Pointer to a structure containing the payload for this message (see Section 31.7.2)

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

31.6.5 eCLD_ColourControlCommandMoveToSaturationCommandSend

```
teZCL_Status eCLD_ColourControlCommandMoveToSaturationCommandSend(
        uint8 u8SourceEndPointId,
        uint8 u8DestinationEndPointId,
        tsZCL_Address *psDestinationAddress,
        uint8 *pu8TransactionSequenceNumber,
        tsCLD_ColourControl_MoveToSaturationCommandPayload
 *psPayload);
```

Description

This function sends a Move to Saturation command to instruct a device to move its ‘current saturation’ attribute to a target saturation value in a continuous manner within a given time. The saturation value and transition time are specified in the payload of the command (see Section 31.7.2).

The device receiving this message will generate a callback event on the endpoint on which the Colour Control cluster was registered. The device must first ensure that ‘hue and saturation’ mode is selected by setting the ‘colour mode’ attribute to 0x00, if required. It can then move the ‘current saturation’ value as requested.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is 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.

This function can only be used when the ‘current saturation’ attribute is enabled in the Colour Control cluster.

Parameters

- *u8SourceEndPointId*: Number of the local endpoint through which to send the request. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values
- *u8DestinationEndPointId*: Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP
- *psDestinationAddress*: Pointer to a structure holding the address of the node to which the request is sent
- *pu8TransactionSequenceNumber*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the request
- *psPayload*: Pointer to a structure containing the payload for this message (see Section 31.7.2)

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

31.6.6 eCLD_ColourControlCommandMoveSaturationCommandSend

```
teZCL_Status eCLD_ColourControlCommandMoveSaturationCommandSend(
        uint8 u8SourceEndPointId,
        uint8 u8DestinationEndPointId,
        tsZCL_Address *psDestinationAddress,
        uint8 *pu8TransactionSequenceNumber,
        tsCLD_ColourControl_MoveSaturationCommandPayload
 *psPayload);
```

Description

This function sends a Move Saturation command to instruct a device to move its ‘current saturation’ attribute value in a given direction at a specified rate for an indefinite time. The direction and rate are specified in the payload of the command (see Section 31.7.2).

The command can request that the saturation is moved up or down, or that existing movement is stopped. Once started, the movement will continue until it is stopped. If the current saturation reaches its minimum or maximum value, the movement will automatically stop.

The device receiving this message will generate a callback event on the endpoint on which the Colour Control cluster was registered. The device must first ensure that ‘hue and saturation’ mode is selected by setting the ‘colour mode’ attribute to 0x00, if required. It can then move the ‘current saturation’ value as requested.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is 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.

This function can only be used when the ‘current saturation’ attribute is enabled in the Colour Control cluster.

Parameters

- *u8SourceEndPointId*: Number of the local endpoint through which to send the request. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values
- *u8DestinationEndPointId*: Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP
- *psDestinationAddress*: Pointer to a structure holding the address of the node to which the request is sent
- *pu8TransactionSequenceNumber*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the request
- *psPayload*: Pointer to a structure containing the payload for this message (see Section 31.7.2)

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

31.6.7 eCLD_ColourControlCommandStepSaturationCommandSend

```
teZCL_Status eCLD_ColourControlCommandStepSaturationCommandSend(
        uint8 u8SourceEndPointId,
        uint8 u8DestinationEndPointId,
        tsZCL_Address *psDestinationAddress,
        uint8 *pu8TransactionSequenceNumber,
        tsCLD_ColourControl_StepSaturationCommandPayload
 *psPayload);
```

Description

This function sends a Step Saturation command to instruct a device to increase or decrease its ‘current saturation’ attribute by a specified ‘step’ value in a continuous manner within a given time. The step size, direction and transition time are specified in the payload of the command (see Section 31.7.2).

The device receiving this message will generate a callback event on the endpoint on which the Colour Control cluster was registered. The device must first ensure that ‘hue and saturation’ mode is selected by setting the ‘colour mode’ attribute to 0x00, if required. It can then move the ‘current saturation’ value as requested.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is 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.

This function can only be used when the ‘current saturation’ attribute is enabled in the Colour Control cluster.

Parameters

- *u8SourceEndPointId*: Number of the local endpoint through which to send the request. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values
- *u8DestinationEndPointId*: Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP
- *psDestinationAddress*: Pointer to a structure holding the address of the node to which the request is sent
- *pu8TransactionSequenceNumber*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the request
- *psPayload*: Pointer to a structure containing the payload for this message (see Section 31.7.2)

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

31.6.8 eCLD_ColourControlCommandMoveToHueAndSaturationCommandSend

```
teZCL_Status eCLD_ColourControlCommandMoveToHueCommandSend(
        uint8 u8SourceEndPointId,
        uint8 u8DestinationEndPointId,
        tsZCL_Address *psDestinationAddress,
        uint8 *pu8TransactionSequenceNumber,
        tsCLD_ColourControl_MoveToHueCommandPayload
 *psPayload);
```

Description

This function sends a Move to Hue and Saturation command to instruct a device to move its ‘current hue’ and ‘current saturation’ attributes to target values in a continuous manner within a given time. The hue value, saturation value and transition time are specified in the payload of the command (see Section 31.7.2).

The device receiving this message will generate a callback event on the endpoint on which the Colour Control cluster was registered. The device must first ensure that ‘hue and saturation’ mode is selected by setting the ‘colour mode’ attribute to 0x00, if required. It can then move the ‘current hue’ and ‘current saturation’ values as requested.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is 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.

This function can only be used when the ‘current hue’ and ‘current saturation’ attributes are enabled in the Colour Control cluster.

Parameters

- *u8SourceEndPointId*: Number of the local endpoint through which to send the request. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values
- *u8DestinationEndPointId*: Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP
- *psDestinationAddress*: Pointer to a structure holding the address of the node to which the request is sent
- *pu8TransactionSequenceNumber*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the request
- *psPayload*: Pointer to a structure containing the payload for this message (see Section 31.7.2)

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

31.6.9 eCLD_ColourControlCommandMoveToColourCommandSend

```
teZCL_Status eCLD_ColourControlCommandMoveToColourCommandSend(
        uint8 u8SourceEndPointId,
        uint8 u8DestinationEndPointId,
        tsZCL_Address *psDestinationAddress,
        uint8 *pu8TransactionSequenceNumber,
        tsCLD_ColourControl_MoveToColourCommandPayload
 *psPayload);
```

Description

This function sends a Move to Colour command to instruct a device to move its ‘current x’ and ‘current y’ attributes to target values in a continuous manner within a given time (where x and y are the chromaticities from the CIE xyY colour space). The x-value, y-value and transition time are specified in the payload of the command (see Section 31.7.2).

The device receiving this message will generate a callback event on the endpoint on which the Colour Control cluster was registered. The device must first ensure that ‘chromaticities x and y’ mode is selected by setting the ‘colour mode’ attribute to 0x01, if required. It can then move the ‘current x’ and ‘current y’ values as requested.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is 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.

This function can only be used when the ‘current x’ and ‘current y’ attributes are enabled in the Colour Control cluster.

Parameters

- *u8SourceEndPointId*: Number of the local endpoint through which to send the request. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values
- *u8DestinationEndPointId*: Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP
- *psDestinationAddress*: Pointer to a structure holding the address of the node to which the request is sent
- *pu8TransactionSequenceNumber*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the request
- *psPayload*: Pointer to a structure containing the payload for this message (see Section 31.7.2)

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

31.6.10 eCLD_ColourControlCommandMoveColourCommandSend

```
teZCL_Status eCLD_ColourControlCommandMoveColourCommandSend(
        uint8 u8SourceEndPointId,
        uint8 u8DestinationEndPointId,
        tsZCL_Address *psDestinationAddress,
        uint8 *pu8TransactionSequenceNumber,
        tsCLD_ColourControl_MoveColourCommandPayload
 *psPayload);
```

Description

This function sends a Move Colour command to instruct a device to move its ‘current x’ and ‘current y’ attribute values at a specified rate for each attribute for an indefinite time (where x and y are the chromaticities from the CIE xyY colour space). The rates are specified in the payload of the command (see Section 31.7.2 and each rate can be positive (increase) or negative (decrease).

Once started, the movement will continue until it is stopped. The movement can be stopped by calling this function with both rates set to zero. The movement will be automatically stopped when either of the attributes reaches its minimum of maximum value.

The device receiving this message will generate a callback event on the endpoint on which the Colour Control cluster was registered. The device must first ensure that ‘chromaticities x and y’ mode is selected by setting the ‘colour mode’ attribute to 0x01, if required. It can then move the ‘current x’ and ‘current y’ values as requested.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is 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.

This function can only be used when the ‘current x’ and ‘current y’ values attributes are enabled in the Colour Control cluster.

Parameters

- *u8SourceEndPointId*: Number of the local endpoint through which to send the request. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values
- *u8DestinationEndPointId*: Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP
- *psDestinationAddress*: Pointer to a structure holding the address of the node to which the request is sent
- *pu8TransactionSequenceNumber*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the request
- *psPayload*: Pointer to a structure containing the payload for this message (see Section 31.7.2)

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

31.6.11 eCLD_ColourControlCommandStepColourCommandSend

```
teZCL_Status eCLD_ColourControlCommandStepColourCommandSend(
        uint8 u8SourceEndPointId,
        uint8 u8DestinationEndPointId,
        tsZCL_Address *psDestinationAddress,
        uint8 *pu8TransactionSequenceNumber,
        tsCLD_ColourControl_StepColourCommandPayload
```

```
  *psPayload);
```

Description

This function sends a Step Colour command to instruct a device to change its ‘current x’ and ‘current y’ attribute values by a specified ‘step’ value for each attribute in a continuous manner within a given time (where x and y are the chromaticities from the CIE xyY colour space). The step sizes and transition time are specified in the payload of the command (see Section 31.7.2), and each step size can be positive (increase) or negative (decrease).

The device receiving this message will generate a callback event on the endpoint on which the Colour Control cluster was registered. The device must first ensure that ‘chromaticities x and y’ mode is selected by setting the ‘colour mode’ attribute to 0x01, if required. It can then move the ‘current x’ and ‘current y’ values as requested.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is 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.

This function can only be used when the ‘current x’ and ‘current y’ values attributes are enabled in the Colour Control cluster.

Parameters

- *u8SourceEndPointId*: Number of the local endpoint through which to send the request. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values
- *u8DestinationEndPointId*: Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP
- *psDestinationAddress*: Pointer to a structure holding the address of the node to which the request is sent
- *pu8TransactionSequenceNumber*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the request
- *psPayload*: Pointer to a structure containing the payload for this message (see Section 31.7.2)

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

31.6.12 eCLD_ColourControlCommandEnhancedMoveToHueCommandSend

```
teZCL_Status eCLD_ColourControlCommandEnhancedMoveToHueCommandSend(
        uint8 u8SourceEndPointId,
        uint8 u8DestinationEndPointId,
        tsZCL_Address *psDestinationAddress,
        uint8 *pu8TransactionSequenceNumber,
```

```
        tsCLD_ColourControl_EnhancedMoveToHueCommandPayload
*psPayload);
```

Description

This function sends an Enhanced Move to Hue command to instruct a device to move its ‘enhanced current hue’ attribute to a target hue value in a continuous manner within a given time. The enhanced hue value, direction and transition time are specified in the payload of the command (see Section 31.7.2). The ‘current hue’ attribute is also moved to a value based on the target ‘enhanced current hue’ value.

Since the possible hues are represented on a closed boundary, the target hue can be reached by moving the attribute value in either direction, up or down (the attribute value wraps around). Options are also provided for ‘shortest route’ and ‘longest route’ around the boundary.

The device receiving this message will generate a callback event on the endpoint on which the Colour Control cluster was registered. The device must first ensure that ‘hue and saturation’ mode is selected by setting the ‘colour mode’ attribute to 0x00 and that ‘enhanced hue and saturation’ mode is selected by setting the ‘enhanced colour mode’ attribute to 0x03, if required. It can then move the ‘enhanced current hue’ value as requested.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is 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.

This function can only be used when the ‘enhanced current hue’ attribute is enabled in the Colour Control cluster.

Parameters

- *u8SourceEndPointId*: Number of the local endpoint through which to send the request. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values
- *u8DestinationEndPointId*: Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP
- *psDestinationAddress*: Pointer to a structure holding the address of the node to which the request is sent
- *pu8TransactionSequenceNumber*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the request
- *psPayload*: Pointer to a structure containing the payload for this message (see Section 31.7.2)

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

31.6.13 eCLD_ColourControlCommandEnhancedMoveHueCommandSend

```
teZCL_Status eCLD_ColourControlCommandEnhancedMoveHueCommandSend(
        uint8 u8SourceEndPointId,
        uint8 u8DestinationEndPointId,
        tsZCL_Address *psDestinationAddress,
        uint8 *pu8TransactionSequenceNumber,
        tsCLD_ColourControl_EnhancedMoveHueCommandPayload
 *psPayload);
```

Description

This function sends an Enhanced Move Hue command to instruct a device to move its ‘enhanced current hue’ attribute value in a given direction at a specified rate for an indefinite time. The direction and rate are specified in the payload of the command (see Section 31.7.2). The ‘current hue’ attribute is also moved through values based on the ‘enhanced current hue’ value.

The command can request that the hue is moved up or down, or that existing movement is stopped. Since the possible hues are represented on a closed boundary, the movement is cyclic (the attribute value wraps around). Once started, the movement will continue until it is stopped.

The device receiving this message will generate a callback event on the endpoint on which the Colour Control cluster was registered. The device must first ensure that ‘hue and saturation’ mode is selected by setting the ‘colour mode’ attribute to 0x00 and that ‘enhanced hue and saturation’ mode is selected by setting the ‘enhanced colour mode’ attribute to 0x03, if required. It can then move the ‘enhanced current hue’ value as requested.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is 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.

This function can only be used when the ‘enhanced current hue’ attribute is enabled in the Colour Control cluster.

Parameters

- *u8SourceEndPointId*: Number of the local endpoint through which to send the request. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values
- *u8DestinationEndPointId*: Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP
- *psDestinationAddress*: Pointer to a structure holding the address of the node to which the request is sent
- *pu8TransactionSequenceNumber*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the request
- *psPayload*: Pointer to a structure containing the payload for this message (see Section 31.7.2)

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL

- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

31.6.14 eCLD_ColourControlCommandEnhancedStepHueCommandSend

```
teZCL_Status eCLD_ColourControlCommandEnhancedStepHueCommandSend(
        uint8 u8SourceEndPointId,
        uint8 u8DestinationEndPointId,
        tsZCL_Address *psDestinationAddress,
        uint8 *pu8TransactionSequenceNumber,
        tsCLD_ColourControl_EnhancedStepHueCommandPayload
 *psPayload);
```

Description

This function sends an Enhanced Step Hue command to instruct a device to increase or decrease its ‘enhanced current hue’ attribute by a specified ‘step’ value in a continuous manner within a given time. The step size, direction and transition time are specified in the payload of the command (see Section 31.7.2). The ‘current hue’ attribute is also moved through values based on the ‘enhanced current hue’ value.

The device receiving this message will generate a callback event on the endpoint on which the Colour Control cluster was registered. The device must first ensure that ‘hue and saturation’ mode is selected by setting the ‘colour mode’ attribute to 0x00 and that ‘enhanced hue and saturation’ mode is selected by setting the ‘enhanced colour mode’ attribute to 0x03, if required. It can then move the ‘enhanced current hue’ value as requested.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is 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.

This function can only be used when the ‘enhanced current hue’ attribute is enabled in the Colour Control cluster.

Parameters

- *u8SourceEndPointId*: Number of the local endpoint through which to send the request. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values
- *u8DestinationEndPointId*: Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP
- *psDestinationAddress*: Pointer to a structure holding the address of the node to which the request is sent
- *pu8TransactionSequenceNumber*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the request
- *psPayload*: Pointer to a structure containing the payload for this message (see Section 31.7.2)

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND

- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

31.6.15 eCLD_ColourControlCommandEnhancedMoveToHueAndSaturationCommandSend

```
teZCL_Status eCLD_ColourControlCommandEnhancedMoveToHueAndSaturationCommandSend(
        uint8 u8SourceEndPointId,
        uint8 u8DestinationEndPointId,
        tsZCL_Address *psDestinationAddress,
        uint8 *pu8TransactionSequenceNumber,
                                tsCLD_ColourControl_EnhancedMoveToHueAndSaturation
 CommandPayload *psPayload);
```

Description

This function sends an Enhanced Move to Hue and Saturation command to instruct a device to move its ‘enhanced current hue’ and ‘current saturation’ attributes to target values in a continuous manner within a given time. The enhanced hue value, saturation value and transition time are specified in the payload of the command (see Section 31.7.2). The ‘current hue’ attribute is also moved to a value based on the target ‘enhanced current hue’ value.

The device receiving this message will generate a callback event on the endpoint on which the Colour Control cluster was registered. The device must first ensure that ‘hue and saturation’ mode is selected by setting the ‘colour mode’ attribute to 0x00 and that ‘enhanced hue and saturation’ mode is selected by setting the ‘enhanced colour mode’ attribute to 0x03, if required. It can then move the ‘enhanced current hue’ and ‘current saturation’ values as requested.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is 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.

This function can only be used when the ‘enhanced current hue’ and ‘current saturation’ attributes are enabled in the Colour Control cluster.

Parameters

- *u8SourceEndPointId*: Number of the local endpoint through which to send the request. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values
- *u8DestinationEndPointId*: Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP
- *psDestinationAddress*: Pointer to a structure holding the address of the node to which the request is sent
- *pu8TransactionSequenceNumber*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the request
- *psPayload*: Pointer to a structure containing the payload for this message (see Section 31.7.2)

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE

- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

31.6.16 eCLD_ColourControlCommandColourLoopSetCommandSend

```
teZCL_Status          eCLD_ColourControlCommandColourLoopSetCommandSend(
        uint8 u8SourceEndPointId,
        uint8 u8DestinationEndPointId,
        tsZCL_Address *psDestinationAddress,
        uint8 *pu8TransactionSequenceNumber,
        tsCLD_ColourControl_ColourLoopSetCommandPayload
         *psPayload);
```

Description

This function sends a Colour Loop Set command to instruct a device to configure the movement of the ‘enhanced current hue’ attribute value around the colour loop corresponding to the CIE colour ‘triangle’. The configured movement can be started in either direction and for a specific duration. The start hue, direction and duration are specified in the payload of the command (see Section 31.7.2). The ‘current hue’ attribute is also moved through values based on the ‘enhanced current hue’ value.

The function can also be used to stop existing movement around the colour loop.

The device receiving this message will generate a callback event on the endpoint on which the Colour Control cluster was registered. The device must first ensure that ‘hue and saturation’ mode is selected by setting the ‘colour mode’ attribute to 0x00 and that ‘enhanced hue and saturation’ mode is selected by setting the ‘enhanced colour mode’ attribute to 0x03, if required. It can then move the ‘enhanced current hue’ value as requested.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is 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.

This function can only be used when the ‘enhanced current hue’ attribute is enabled in the Colour Control cluster.

Parameters

- *u8SourceEndPointId*: Number of the local endpoint through which to send the request. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values
- *u8DestinationEndPointId*: Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP
- *psDestinationAddress*: Pointer to a structure holding the address of the node to which the request is sent
- *pu8TransactionSequenceNumber*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the request
- *psPayload*: Pointer to a structure containing the payload for this message (see Section 31.7.2)

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

31.6.17 eCLD_ColourControlCommandStopMoveStepCommandSend

```
teZCL_Status eCLD_ColourControlCommandStopMoveStepCommandSend(
        uint8 u8SourceEndPointId,
        uint8 u8DestinationEndPointId,
        tsZCL_Address *psDestinationAddress,
        uint8 *pu8TransactionSequenceNumber,
         tsCLD_ColourControl_StopMoveStepCommandPayload
 *psPayload);
```

Description

This function sends a Stop Move Step command to instruct a device to stop a ‘Move to’, ‘Move’ or ‘Step’ command that is currently in progress.

The device receiving this message will generate a callback event on the endpoint on which the Colour Control cluster was registered, and stop the current action.

The ‘current hue’, ‘enhanced current hue’ and ‘current saturation’ attributes will subsequently keep the values they have when the current action is stopped.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is 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.

This function can only be used when the ‘enhanced current hue’ attribute is enabled in the Colour Control cluster.

Parameters

- *u8SourceEndPointId*: Number of the local endpoint through which to send the request. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values
- *u8DestinationEndPointId*: Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP
- *psDestinationAddress*: Pointer to a structure holding the address of the node to which the request is sent
- *pu8TransactionSequenceNumber*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the request
- *psPayload* : Pointer to a structure containing the payload for this message (see Section 31.7.2)

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

31.6.18 eCLD_ColourControlCommandMoveToColourTemperatureCommandSend

```
teZCL_Status eCLD_ColourControlCommandMoveToColourTemperatureCommandSend(
        uint8 u8SourceEndPointId,
        uint8 u8DestinationEndPointId,
        tsZCL_Address *psDestinationAddress,
        uint8 *pu8TransactionSequenceNumber,
        tsCLD_ColourControl_MoveToColourTemperatureCommandPayload
 *psPayload);
```

Description

This function sends a Move to Colour Temperature command to instruct a device to move its ‘mired colour temperature’ attribute to a target value in a continuous manner within a given time. The attribute value is a scaled reciprocal of colour temperature, as indicated in Section 31.5.4. The target attribute value, direction and transition time are specified in the payload of the command (see Section 31.7.2).

The movement through colour space will follow the ‘Black Body Line'.

The device receiving this message will generate a callback event on the endpoint on which the Colour Control cluster was registered. The device must first ensure that ‘colour temperature’ mode is selected by setting the ‘colour mode’ attribute to 0x02, if required. It can then move the ‘mired colour temperature’ value as requested.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is 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.

This function can only be used when the ‘mired colour temperature’ attribute is enabled in the Colour Control cluster.

Parameters

- *u8SourceEndPointId*: Number of the local endpoint through which to send the request. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values
- *u8DestinationEndPointId*: Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP
- *psDestinationAddress*: Pointer to a structure holding the address of the node to which the request is sent
- *pu8TransactionSequenceNumber*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the request
- *psPayload*: Pointer to a structure containing the payload for this message (see Section 31.7.2)

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

31.6.19 eCLD_ColourControlCommandMoveColourTemperatureCommandSend

```
teZCL_Status eCLD_ColourControlCommandMoveColourTemperatureCommandSend(
        uint8 u8SourceEndPointId,
        uint8 u8DestinationEndPointId,
        tsZCL_Address *psDestinationAddress,
        uint8 *pu8TransactionSequenceNumber,
        tsCLD_ColourControl_MoveColourTemperatureCommandPayload
 *psPayload);
```

Description

This function sends a Move Colour Temperature command to instruct a device to move its ‘mired colour temperature’ attribute value in a given direction at a specified rate. The attribute value is a scaled reciprocal of colour temperature, as indicated in Section 31.5.4. The direction and rate are specified in the payload of the command (see Section 31.7.2). Maximum and minimum attribute values for the movement are also specified in the payload.

The command can request that the attribute value is moved up or down, or that existing movement is stopped. Once started, the movement will automatically stop when the attribute value reaches the specified maximum or minimum.

The movement through colour space will follow the ‘Black Body Line'.

The device receiving this message will generate a callback event on the endpoint on which the Colour Control cluster was registered. The device must first ensure that ‘colour temperature’ mode is selected by setting the ‘colour mode’ attribute to 0x02, if required. It can then move the ‘mired colour temperature’ value as requested.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is 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.

This function can only be used when the ‘mired colour temperature’ attribute is enabled in the Colour Control cluster, as well as the ‘mired colour temperature maximum’ and ‘mired colour temperature minimum’ attributes.

Parameters

- *u8SourceEndPointId*: Number of the local endpoint through which to send the request. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values
- *u8DestinationEndPointId*: Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP

- *psDestinationAddress*: Pointer to a structure holding the address of the node to which the request is sent
- *pu8TransactionSequenceNumber*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the request
- *psPayload*: Pointer to a structure containing the payload for this message (see Section 31.7.2)

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

31.6.20 eCLD_ColourControlCommandStepColourTemperatureCommandSend

```
teZCL_Status eCLD_ColourControlCommandStepColourTemperatureCommandSend(
        uint8 u8SourceEndPointId,
        uint8 u8DestinationEndPointId,
        tsZCL_Address *psDestinationAddress,
        uint8 *pu8TransactionSequenceNumber,
        tsCLD_ColourControl_StepColourTemperatureCommandPayload
 *psPayload);
```

Description

This function sends a Step Colour Temperature command to instruct a device to increase or decrease its ‘mired colour temperature’ attribute by a specified ‘step’ value in a continuous manner within a given time. The attribute value is a scaled reciprocal of colour temperature, as indicated in Section 31.5.4. The step size, direction and transition time are specified in the payload of the command (see Section 31.7.2). Maximum and minimum attribute values for the movement are also specified in the payload.

The command can request that the attribute value is moved up or down. If this value reaches the specified maximum or minimum before the required change has been achieved, the movement will automatically stop.

The movement through colour space will follow the ‘Black Body Line'.

The device receiving this message will generate a callback event on the endpoint on which the Colour Control cluster was registered. The device must first ensure that ‘colour temperature’ mode is selected by setting the ‘colour mode’ attribute to 0x02, if required. It can then move the ‘mired colour temperature’ value as requested.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is 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.

This function can only be used when the ‘mired colour temperature’ attribute is enabled in the Colour Control cluster, as well as the ‘mired colour temperature maximum’ and ‘mired colour temperature minimum’ attributes.

Parameters

- *u8SourceEndPointId*: Number of the local endpoint through which to send the request. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values
- *u8DestinationEndPointId*: Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP
- *psDestinationAddress*: Pointer to a structure holding the address of the node to which the request is sent
- *pu8TransactionSequenceNumber*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the request
- *psPayload*: Pointer to a structure containing the payload for this message (see Section 31.7.2)

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

31.6.21 eCLD_ColourControl_GetRGB

```
teZCL_Status eCLD_ColourControl_GetRGB(
        uint8 u8SourceEndPointId,
        uint8 *pu8Red,
        uint8 *pu8Green,
        uint8 *pu8Blue);
```

Description

This function obtains the current colour of the device on the specified (local) endpoint in terms of the Red (R), Green (G) and Blue (B) components.

Parameters

- *u8SourceEndPointId*: Number of local endpoint on which the device resides
- *pu8Red*: Pointer to a location to receive the red value, in the range 0-255
- *pu8Green*: Pointer to a location to receive the green value, in the range 0-255
- *pu8Blue*: Pointer to a location to receive the blue value, in the range 0-255

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN

- E_ZCL_ERR_CLUSTER_NOT_FOUND

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

31.7 Structures

31.7.1 Custom Data Structure

The Colour Control cluster requires extra storage space to be allocated for use by internal functions. The structure definition for this storage is shown below:

```
typedef struct
{
    teCLD_ColourControl_ColourMode     eColourMode;
    uint16                             u16CurrentHue;
    tsCLD_ColourControl_Transition     sTransition;
    /* Matrices for XYZ <> RGB conversions */
    float                              afXYZ2RGB[3][3];
    float                              afRGB2XYZ[3][3];
    tsZCL_ReceiveEventAddress          sReceiveEventAddress;
    tsZCL_CallBackEvent                sCustomCallBackEvent;
    tsCLD_ColourControlCallBackMessage sCallBackMessage;
} tsCLD_ColourControlCustomDataStructure;
```

The fields are for internal use and no knowledge of them is required.

31.7.2 Custom Command Payloads

The following structures contain the payloads for the Colour Control cluster custom commands.

Move to Hue Command Payload

```
typedef struct
{
    uint8                           u8Hue;
    teCLD_ColourControl_Direction   eDirection;
    uint16                          u16TransitionTime;
} tsCLD_ColourControl_MoveToHueCommandPayload;
```

where:

- `u8Hue` is the target hue value.
- `eDirection` indicates the direction/path of the change in hue:

eDirection	Direction/Path
0x00	Shortest path
0x01	Longest path
0x02	Up
0x03	Down
0x04 – 0xFF	Reserved

- `u16TransitionTime` is the time period, in tenths of a second, over which the change in hue should be implemented.

Move Hue Command Payload

```
typedef struct
{
    teCLD_ColourControl_MoveMode    eMode;
    uint8                           u8Rate;
    zbmap8                          u8OptionsMask;
    zbmap8                          u8OptionsOverride;
} tsCLD_ColourControl_MoveHueCommandPayload;
```

where:

- `eMode` indicates the required action and/or direction of the change in hue:

eMode	Action/Direction
0x00	Stop existing movement in hue
0x01	Start increasing hue
0x02	Reserved
0x03	Start decreasing hue
0x04 – 0xFF	Reserved

- `u8Rate` is the required rate of movement in hue steps per second (a step is one unit of hue for the device).
- `OptionsMask` and `OptionsOverride` must be either both present or both not present. These fields are used in creating a temporary Options bitmap from the `u8Options` attribute. Each bit of the `u8Options` attribute is carried across to the temporary Options bitmap unless the corresponding bit of `OptionsMask` is set (to 1). In this case, the corresponding bit of `OptionsOverride` is used in the temporary Options bitmap instead.

Step Hue Command Payload

```
typedef struct
{
    teCLD_ColourControl_StepMode    eMode;
    uint8                           u8StepSize;
    uint8                           u8TransitionTime;
    zbmap8                          u8OptionsMask;
    zbmap8                          u8OptionsOverride;
} tsCLD_ColourControl_StepHueCommandPayload;
```

where:

- `eMode` indicates the required direction of the change in hue:

eMode	Action/Direction
0x00	Reserved
0x01	Increase hue
0x02	Reserved
0x03	Decrease hue

eMode	Action/Direction
0x04 – 0xFF	Reserved

- `u8StepSize` is the amount by which the hue is to be changed (increased or decreased), in units of hue for the device.
- `u8TransitionTime` is the time period, in tenths of a second, over which the change in hue should be implemented.
- `OptionsMask` and `OptionsOverride` must be either both present or both not present. These fields are used in creating a temporary Options bitmap from the `u8Options` attribute. Each bit of the `u8Options` attribute is carried across to the temporary Options bitmap unless the corresponding bit of `OptionsMask` is set (to 1). In this case, the corresponding bit of `OptionsOverride` is used in the temporary Options bitmap instead.

Move To Saturation Command Payload

```
typedef struct
{
    uint8                           u8Saturation;
    uint16                          u16TransitionTime;
    zbmap8                          u8OptionsMask;
    zbmap8                          u8OptionsOverride;
} tsCLD_ColourControl_MoveToSaturationCommandPayload;
```

where:

- `u8Saturation` is the target saturation value.
- `u16TransitionTime` is the time period, in tenths of a second, over which the change in saturation should be implemented.
- `OptionsMask` and `OptionsOverride` must be either both present or both not present. These fields are used in creating a temporary Options bitmap from the `u8Options` attribute. Each bit of the `u8Options` attribute is carried across to the temporary Options bitmap unless the corresponding bit of `OptionsMask` is set (to 1). In this case, the corresponding bit of `OptionsOverride` is used in the temporary Options bitmap instead.

Move Saturation Command Payload

```
typedef struct
{
    teCLD_ColourControl_MoveMode    eMode;
    uint8                           u8Rate;
    zbmap8                          u8OptionsMask;
    zbmap8                          u8OptionsOverride;
} tsCLD_ColourControl_MoveSaturationCommandPayload;
```

where:

- `eMode` indicates the required action and/or direction of the change in saturation:

eMode	Action/Direction
0x00	Stop existing movement in hue
0x01	Start increasing saturation
0x02	Reserved

eMode	Action/Direction
0x03	Start decreasing saturation
0x04 – 0xFF	Reserved

- `u8Rate` is the required rate of movement in saturation steps per second (a step is one unit of saturation for the device).
- `OptionsMask` and `OptionsOverride` must be either both present or both not present. These fields are used in creating a temporary Options bitmap from the `u8Options` attribute. Each bit of the `u8Options` attribute is carried across to the temporary Options bitmap unless the corresponding bit of `OptionsMask` is set (to 1). In this case, the corresponding bit of `OptionsOverride` is used in the temporary Options bitmap instead.

Step Saturation Command Payload

```
typedef struct
{
    teCLD_ColourControl_StepMode    eMode;
    uint8                           u8StepSize;
    uint8                           u8TransitionTime;
    zbmap8                          u8OptionsMask;
    zbmap8                          u8OptionsOverride;
} tsCLD_ColourControl_StepSaturationCommandPayload;
```

where:

- `eMode` indicates the required direction of the change in saturation:

eMode	Action/Direction
0x00	Reserved
0x01	Increase saturation
0x02	Reserved
0x03	Decrease saturation
0x04 – 0xFF	Reserved

- `u8StepSize` is the amount by which the saturation is to be changed (increased or decreased), in units of saturation for the device.
- `u8TransitionTime` is the time period, in tenths of a second, over which the change in hue should be implemented.
- `OptionsMask` and `OptionsOverride` must be either both present or both not present. These fields are used in creating a temporary Options bitmap from the `u8Options` attribute. Each bit of the `u8Options` attribute is carried across to the temporary Options bitmap unless the corresponding bit of `OptionsMask` is set (to 1). In this case, the corresponding bit of `OptionsOverride` is used in the temporary Options bitmap instead.

Move To Hue And Saturation Command Payload

```
typedef struct
{
    uint8                           u8Hue;
    uint8                           u8Saturation;
    uint16                          u16TransitionTime;
```

```
    zbmap8                          u8OptionsMask;
    zbmap8                          u8OptionsOverride;
} tsCLD_ColourControl_MoveToHueAndSaturationCommandPayload;
```

where:

- `u8Hue` is the target hue value.
- `u8Saturation` is the target saturation value.
- `16TransitionTime` is the time period, in tenths of a second, over which the change in hue and saturation should be implemented.
- `OptionsMask` and `OptionsOverride` must be either both present or both not present. These fields are used in creating a temporary Options bitmap from the `u8Options` attribute. Each bit of the `u8Options` attribute is carried across to the temporary Options bitmap unless the corresponding bit of `OptionsMask` is set (to 1). In this case, the corresponding bit of `OptionsOverride` is used in the temporary Options bitmap instead.

Move To Colour Command Payload

```
typedef struct
{
    uint16                          u16ColourX;
    uint16                          u16ColourY;
    uint16                          u16TransitionTime;
    zbmap8                          u8OptionsMask;
    zbmap8                          u8OptionsOverride;
} tsCLD_ColourControl_MoveToColourCommandPayload;
```

where:

- `u16ColourX` is the target x-chromaticity in the CIE xyY colour space
- `u16ColourY` is the target y-chromaticity in the CIE xyY colour space
- `u16TransitionTime` is the time period, in tenths of a second, over which the colour change should be implemented.
- `OptionsMask` and `OptionsOverride` must be either both present or both not present. These fields are used in creating a temporary Options bitmap from the `u8Options` attribute. Each bit of the `u8Options` attribute is carried across to the temporary Options bitmap unless the corresponding bit of `OptionsMask` is set (to 1). In this case, the corresponding bit of `OptionsOverride` is used in the temporary Options bitmap instead.

Move Colour Command Payload

```
typedef struct
{
    int16                           i16RateX;
    int16                           i16RateY;
    zbmap8                          u8OptionsMask;
    zbmap8                          u8OptionsOverride;
} tsCLD_ColourControl_MoveColourCommandPayload;
```

where:

- `i16RateX` is the required rate of movement of x-chromaticity in the CIE xyY colour space, in steps per second (a step is one unit of x-chromaticity for the device).
- `i16RateY` is the required rate of movement of y-chromaticity in the CIE xyY colour space, in steps per second (a step is one unit of y-chromaticity for the device).

- `OptionsMask` and `OptionsOverride` must be either both present or both not present. These fields are used in creating a temporary Options bitmap from the `u8Options` attribute. Each bit of the `u8Options` attribute is carried across to the temporary Options bitmap unless the corresponding bit of `OptionsMask` is set (to 1). In this case, the corresponding bit of `OptionsOverride` is used in the temporary Options bitmap instead.

Step Colour Command Payload

```
typedef struct
{
    int16                           i16StepX;
    int16                           i16StepY;
    uint16                          u16TransitionTime;
    zbmap8                          u8OptionsMask;
    zbmap8                          u8OptionsOverride;
} tsCLD_ColourControl_StepColourCommandPayload;
```

where:

- `i16StepX` is the amount by which the x-chromaticity in the CIE xyY colour space is to be changed (increased or decreased), in units of x-chromaticity for the device.
- `i16StepY` is the amount by which the y-chromaticity in the CIE xyY colour space is to be changed (increased or decreased), in units of y-chromaticity for the device.
- `u16TransitionTime` is the time period, in tenths of a second, over which the colour change should be implemented.
- `OptionsMask` and `OptionsOverride` must be either both present or both not present. These fields are used in creating a temporary Options bitmap from the `u8Options` attribute. Each bit of the `u8Options` attribute is carried across to the temporary Options bitmap unless the corresponding bit of `OptionsMask` is set (to 1). In this case, the corresponding bit of `OptionsOverride` is used in the temporary Options bitmap instead.

Move To Colour Temperature Command Payload

```
typedef struct
{
    uint16                          u16ColourTemperatureMired;
    uint16                          u16TransitionTime;
    zbmap8                          u8OptionsMask;
    zbmap8                          u8OptionsOverride;
} tsCLD_ColourControl_MoveToColourTemperatureCommandPayload;
```

where:

- `u16ColourTemperatureMired` is the target value of the mired colour temperature attribute `u16ColourTemperatureMired` (this value is a scaled reciprocal of colour temperature - for details, refer to the attribute description in Section 31.2).
- `u16TransitionTime` is the time period, in tenths of a second, over which the change in colour temperature should be implemented.
- `OptionsMask` and `OptionsOverride` must be either both present or both not present. These fields are used in creating a temporary Options bitmap from the `u8Options` attribute. Each bit of the `u8Options` attribute is carried across to the temporary Options bitmap unless the corresponding bit of `OptionsMask` is set (to 1). In this case, the corresponding bit of `OptionsOverride` is used in the temporary Options bitmap instead.

Move Colour Temperature Command Payload

```
typedef struct
{
    teCLD_ColourControl_MoveMode     eMode;
    uint16                           u16Rate;
    uint16                           u16ColourTemperatureMiredMin;
    uint16                           u16ColourTemperatureMiredMax;
    zbmap8                           u8OptionsMask;
    zbmap8                           u8OptionsOverride;
} tsCLD_ColourControl_MoveColourTemperatureCommandPayload;
```

where:

- `eMode` indicates the required action and/or direction of the change in the mired colour temperature attribute value:

eMode	Action/Direction
0x00	Stop existing movement in colour temperature
0x01	Start increasing mired colour temperature attribute value
0x02	Reserved
0x03	Start decreasing mired colour temperature attribute value
0x04 – 0xFF	Reserved

- `u16Rate` is the required rate of movement in mired colour temperature steps per second (a step is one unit of the mired colour temperature attribute).
- `u16ColourTemperatureMiredMin` is the lower limit for the mired colour temperature attribute during the operation resulting from this command.
- `u16ColourTemperatureMiredMax` is the upper limit for the mired colour temperature attribute during the operation resulting from this command.
- `OptionsMask` and `OptionsOverride` must be either both present or both not present. These fields are used in creating a temporary Options bitmap from the `u8Options` attribute. Each bit of the `u8Options` attribute is carried across to the temporary Options bitmap unless the corresponding bit of `OptionsMask` is set (to 1). In this case, the corresponding bit of `OptionsOverride` is used in the temporary Options bitmap instead.

Step Colour Temperature Command Payload

```
typedef struct
{
    teCLD_ColourControl_StepMode     eMode;
    uint16                           u16StepSize;
    uint16                           u16TransitionTime;
    uint16                           u16ColourTemperatureMiredMin;
    uint16                           u16ColourTemperatureMiredMax;
    zbmap8                           u8OptionsMask;
    zbmap8                           u8OptionsOverride;
} tsCLD_ColourControl_StepColourTemperatureCommandPayload;
```

where:

- `eMode` indicates the required direction of the change in the mired colour temperature attribute value:

eMode	Action/Direction
0x00	Reserved
0x01	Increase mired colour temperature attribute value
0x02	Reserved
0x03	Decrease mired colour temperature attribute value
0x04 – 0xFF	Reserved

- `u16StepSize` is the amount by which the mired colour temperature attribute is to be changed (increased or decreased).
- `u16TransitionTime` is the time period, in tenths of a second, over which the change in the mired colour temperature attribute should be implemented.
- `u16ColourTemperatureMiredMin` is the lower limit for the mired colour temperature attribute during the operation resulting from this command.
- `u16ColourTemperatureMiredMax` is the upper limit for the mired colour temperature attribute during the operation resulting from this command.
- `OptionsMask` and `OptionsOverride` must be either both present or both not present. These fields are used in creating a temporary Options bitmap from the `u8Options` attribute. Each bit of the `u8Options` attribute is carried across to the temporary Options bitmap unless the corresponding bit of `OptionsMask` is set (to 1). In this case, the corresponding bit of `OptionsOverride` is used in the temporary Options bitmap instead.

Enhanced Move To Hue Command Payload

```
typedef struct
{
    uint16                              u16EnhancedHue;
    teCLD_ColourControl_Direction       eDirection;
    uint16                              u16TransitionTime;
    zbmap8                              u8OptionsMask;
    zbmap8                              u8OptionsOverride;
} tsCLD_ColourControl_EnhancedMoveToHueCommandPayload;
```

where:

- `u16EnhancedHue` is the target ‘enhanced’ hue value in terms of a step around the CIE colour ‘triangle’ - for the format, refer to the description of the attribute `u16EnhancedCurrentHue` in Section 31.2.
- `eDirection` indicates the direction/path of the change in hue:

eDirection	Direction/Path
0x00	Shortest path
0x01	Longest path
0x02	Up
0x03	Down
0x04 – 0xFF	Reserved

- `u16TransitionTime` is the time period, in tenths of a second, over which the change in hue should be implemented.
- `OptionsMask` and `OptionsOverride` must be either both present or both not present. These fields are used in creating a temporary Options bitmap from the `u8Options` attribute. Each bit of the `u8Options`

attribute is carried across to the temporary Options bitmap unless the corresponding bit of `OptionsMask` is set (to 1). In this case, the corresponding bit of `OptionsOverride` is used in the temporary Options bitmap instead.

Enhanced Move Hue Command Payload

```
typedef struct
{
    teCLD_ColourControl_MoveMode        eMode;
    uint16                              u16Rate;
    zbmap8                              u8OptionsMask;
    zbmap8                              u8OptionsOverride;
} tsCLD_ColourControl_EnhancedMoveHueCommandPayload;
```

where:

- `eMode` indicates the required action and/or direction of the change in hue:

eMode	Action/Direction
0x00	Stop existing movement in hue
0x01	Start increase in hue
0x02	Reserved
0x03	Start decrease in hue
0x04 – 0xFF	Reserved

- `u16Rate` is the required rate of movement in ‘enhanced’ hue steps per second (a step is one unit of hue for the device).
- `OptionsMask` and `OptionsOverride` must be either both present or both not present. These fields are used in creating a temporary Options bitmap from the `u8Options` attribute. Each bit of the `u8Options` attribute is carried across to the temporary Options bitmap unless the corresponding bit of `OptionsMask` is set (to 1). In this case, the corresponding bit of `OptionsOverride` is used in the temporary Options bitmap instead.

Enhanced Step Hue Command Payload

```
typedef struct
{
    teCLD_ColourControl_StepMode        eMode;
    uint16                              u16StepSize;
    uint16                              u16TransitionTime;
    zbmap8                              u8OptionsMask;
    zbmap8                              u8OptionsOverride;
} tsCLD_ColourControl_EnhancedStepHueCommandPayload;
```

where:

- `eMode` indicates the required direction of the change in hue:

eMode	Action/Direction
0x00	Reserved
0x01	Increase in hue

eMode	Action/Direction
0x02	Reserved
0x03	Decrease in hue
0x04 – 0xFF	Reserved

- `u16StepSize` is the amount by which the 'enhanced' hue is to be changed (increased or decreased) - for the format, refer to the description of the attribute `u16EnhancedCurrentHue` in Section 31.2.
- `u8TransitionTime` is the time period, in tenths of a second, over which the change in hue should be implemented.
- `OptionsMask` and `OptionsOverride` must be either both present or both not present. These fields are used in creating a temporary Options bitmap from the `u8Options` attribute. Each bit of the `u8Options` attribute is carried across to the temporary Options bitmap unless the corresponding bit of `OptionsMask` is set (to 1). In this case, the corresponding bit of `OptionsOverride` is used in the temporary Options bitmap instead.

Enhanced Move To Hue And Saturation Command Payload

```
typedef struct
{
    uint16                              u16EnhancedHue;
    uint8                               u8Saturation;
    uint16                              u16TransitionTime;
    zbmap8                              u8OptionsMask;
    zbmap8                              u8OptionsOverride;
} tsCLD_ColourControl_EnhancedMoveToHueAndSaturationCommandPayload;
```

where:

- `u16EnhancedHue` is the target 'enhanced' hue value in terms of a step around the CIE colour 'triangle' - for the format, refer to the description of the attribute `u16EnhancedCurrentHue` in Section 31.2.
- `u8Saturation` is the target saturation value.
- `16TransitionTime` is the time period, in tenths of a second, over which the change in hue and saturation should be implemented.
- `OptionsMask` and `OptionsOverride` must be either both present or both not present. These fields are used in creating a temporary Options bitmap from the `u8Options` attribute. Each bit of the `u8Options` attribute is carried across to the temporary Options bitmap unless the corresponding bit of `OptionsMask` is set (to 1). In this case, the corresponding bit of `OptionsOverride` is used in the temporary Options bitmap instead.

Colour Loop Set Command Payload

```
typedef struct
{
    uint8                               u8UpdateFlags;
    teCLD_ColourControl_LoopAction      eAction;
    teCLD_ColourControl_LoopDirection   eDirection;
    uint16                              u16Time;
    uint16                              u16StartHue;
    zbmap8                              u8OptionsMask;
    zbmap8                              u8OptionsOverride;
} tsCLD_ColourControl_ColourLoopSetCommandPayload;
```

where:

- `u8UpdateFlags` is a bitmap indicating which of the other fields of the structure must be set (a bit must be set to '1' to enable the corresponding field, and '0' otherwise):

Bits	Field
0	`eAction`
1	`eDirection`
2	`u16Time`
3	`u16StartHue`
4–7	Reserved

- `eAction` indicates the colour loop action to be taken (if enabled through `u8UpdateFlags`), as one of:

Enumeration	Value	Action
E_CLD_COLOURCONTROL_COLOURLOOP_ACTION_ DEACTIVATE	0x00	Deactivate colour loop
E_CLD_COLOURCONTROL_COLOURLOOP_ACTION_ ACTIVATE_FROM_START	0x01	Activate colour loop from specified start (enhanced) hue value
E_CLD_COLOURCONTROL_COLOURLOOP_ACTION_ ACTIVATE_FROM_CURRENT	0x02	Activate colour from current (enhanced) hue value

- `eDirection` indicates the direction to be taken around the colour loop (if enabled through `u8UpdateFlags`) in terms of the direction of change of `u16EnhancedCurrentHue`:

Enumeration	Value	Direction
E_CLD_COLOURCONTROL_COLOURLOOP_ DIRECTION_DECREMENT	0x00	Decrement current (enhanced) hue value
E_CLD_COLOURCONTROL_COLOURLOOP_ DIRECTION_INCREMENT	0x01	Increment current (enhanced) hue value

- `u16Time` is the period, in seconds, of a full colour loop - that is, the time to cycle all possible values of `u16EnhancedCurrentHue`.
- `u16StartHue` is the value of `u16EnhancedCurrentHue` at which the colour loop is to be started (if enabled through `u8UpdateFlags`).
- `OptionsMask` and `OptionsOverride` must be either both present or both not present. These fields are used in creating a temporary Options bitmap from the `u8Options` attribute. Each bit of the `u8Options` attribute is carried across to the temporary Options bitmap unless the corresponding bit of `OptionsMask` is set (to 1). In this case, the corresponding bit of `OptionsOverride` is used in the temporary Options bitmap instead.

Stop Move Step Command Payload

```
typedef struct
{
    zbmap8                  u8OptionsMask;
```

```
    zbmap8                  u8OptionsOverride;
} tsCLD_ColourControl_StopMoveStepCommandPayload;
```

where `OptionsMask` and `OptionsOverride` must be either both present or both not present. These fields are used in creating a temporary Options bitmap from the `u8Options` attribute. Each bit of the `u8Options` attribute is carried across to the temporary Options bitmap unless the corresponding bit of `OptionsMask` is set (to 1). In this case, the corresponding bit of `OptionsOverride` is used in the temporary Options bitmap instead.

31.8 Enumerations

31.8.1 teCLD_ColourControl_ClusterID

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

```
typedef enum
{
    E_CLD_COLOURCONTROL_ATTR_CURRENT_HUE                = 0x0000,
    E_CLD_COLOURCONTROL_ATTR_CURRENT_SATURATION,
    E_CLD_COLOURCONTROL_ATTR_REMAINING_TIME,
    E_CLD_COLOURCONTROL_ATTR_CURRENT_X,
    E_CLD_COLOURCONTROL_ATTR_CURRENT_Y,
    E_CLD_COLOURCONTROL_ATTR_DRIFT_COMPENSATION,
    E_CLD_COLOURCONTROL_ATTR_COMPENSATION_TEXT,
    E_CLD_COLOURCONTROL_ATTR_COLOUR_TEMPERATURE_MIRED,
    E_CLD_COLOURCONTROL_ATTR_COLOUR_MODE,
    E_CLD_COLOURCONTROL_ATTR_OPTIONS         = 0x000F,
    E_CLD_COLOURCONTROL_ATTR_NUMBER_OF_PRIMARIES     = 0x0010,
    E_CLD_COLOURCONTROL_ATTR_PRIMARY_1_X,
    E_CLD_COLOURCONTROL_ATTR_PRIMARY_1_Y,
    E_CLD_COLOURCONTROL_ATTR_PRIMARY_1_INTENSITY,
    E_CLD_COLOURCONTROL_ATTR_PRIMARY_2_X            = 0x0015,
    E_CLD_COLOURCONTROL_ATTR_PRIMARY_2_Y,
    E_CLD_COLOURCONTROL_ATTR_PRIMARY_2_INTENSITY,
    E_CLD_COLOURCONTROL_ATTR_PRIMARY_3_X            = 0x0019,
    E_CLD_COLOURCONTROL_ATTR_PRIMARY_3_Y,
    E_CLD_COLOURCONTROL_ATTR_PRIMARY_3_INTENSITY,
    E_CLD_COLOURCONTROL_ATTR_PRIMARY_4_X            = 0x0020,
    E_CLD_COLOURCONTROL_ATTR_PRIMARY_4_Y,
    E_CLD_COLOURCONTROL_ATTR_PRIMARY_4_INTENSITY,
    E_CLD_COLOURCONTROL_ATTR_PRIMARY_5_X            = 0x0024,
    E_CLD_COLOURCONTROL_ATTR_PRIMARY_5_Y,
    E_CLD_COLOURCONTROL_ATTR_PRIMARY_5_INTENSITY,
    E_CLD_COLOURCONTROL_ATTR_PRIMARY_6_X            = 0x0028,
    E_CLD_COLOURCONTROL_ATTR_PRIMARY_6_Y,
    E_CLD_COLOURCONTROL_ATTR_PRIMARY_6_INTENSITY,
    E_CLD_COLOURCONTROL_ATTR_WHITE_POINT_X          = 0x0030,
    E_CLD_COLOURCONTROL_ATTR_WHITE_POINT_Y,
    E_CLD_COLOURCONTROL_ATTR_COLOUR_POINT_R_X,
    E_CLD_COLOURCONTROL_ATTR_COLOUR_POINT_R_Y,
    E_CLD_COLOURCONTROL_ATTR_COLOUR_POINT_R_INTENSITY,
    E_CLD_COLOURCONTROL_ATTR_COLOUR_POINT_G_X        = 0x0036,
    E_CLD_COLOURCONTROL_ATTR_COLOUR_POINT_G_Y,
    E_CLD_COLOURCONTROL_ATTR_COLOUR_POINT_G_INTENSITY,
    E_CLD_COLOURCONTROL_ATTR_COLOUR_POINT_B_X        = 0x003A,
    E_CLD_COLOURCONTROL_ATTR_COLOUR_POINT_B_Y,
    E_CLD_COLOURCONTROL_ATTR_COLOUR_POINT_B_INTENSITY,
    E_CLD_COLOURCONTROL_ATTR_ENHANCED_CURRENT_HUE     = 0x4000,
    E_CLD_COLOURCONTROL_ATTR_ENHANCED_COLOUR_MODE,
    E_CLD_COLOURCONTROL_ATTR_COLOUR_LOOP_ACTIVE,
    E_CLD_COLOURCONTROL_ATTR_COLOUR_LOOP_DIRECTION,
    E_CLD_COLOURCONTROL_ATTR_COLOUR_LOOP_TIME,
    E_CLD_COLOURCONTROL_ATTR_COLOUR_LOOP_START_ENHANCED_HUE,
```

```
    E_CLD_COLOURCONTROL_ATTR_COLOUR_LOOP_STORED_ENHANCED_HUE,
    E_CLD_COLOURCONTROL_ATTR_COLOUR_CAPABILITIES       = 0x400a,
    E_CLD_COLOURCONTROL_ATTR_COLOUR_TEMPERATURE_MIRED_PHY_MIN,
    E_CLD_COLOURCONTROL_ATTR_COLOUR_TEMPERATURE_MIRED_PHY_MAX,
    E_CLD_COLOURCONTROL_ATTR_COUPLE_COLOUR_TEMPERATURE_TO_LEVEL_MIN_MIRED,
    E_CLD_COLOURCONTROL_ATTR_STARTUP_COLOUR_TEMPERATURE_MIRED = 0x4010,
} teCLD_ColourControl_ClusterID;
```

31.9 Compile-time Options

To enable the Colour Control cluster in the code to be built, it is necessary to add the following to the **zcl_options.h** file:

```
#define CLD_COLOUR_CONTROL
```

In addition, to include the software for a cluster client or server or both, it is necessary to add one or both of the following to the same file:

```
#define COLOUR_CONTROL_CLIENT
#define COLOUR_CONTROL_SERVER
```

The Colour Cluster cluster attributes reside on the server only. Therefore, attributes should not be enabled in the **zcl_options.h** file for the cluster client.

Optional Attributes

The optional attributes of the Colour Control cluster are enabled/disabled by defining the following in the **zcl_options.h** file:

- For optional attributes from the ‘Colour Information’ attribute set:
  - CLD_COLOURCONTROL_ATTR_REMAINING_TIME
  - CLD_COLOURCONTROL_ATTR_DRIFT_COMPENSATION
  - CLD_COLOURCONTROL_ATTR_COMPENSATION_TEXT
  - CLD_COLOURCONTROL_ATTR_COLOUR_MODE

    Certain attributes from this attribute set are enabled through a ‘Colour Capabilities’ Definition (see below) - these are `u8CurrentHue`, `u8CurrentSaturation` and `u16ColourTemperatureMired`.
- For optional attributes from the ‘Defined Primaries Information’ and ‘Additional Defined Primaries Information’ attribute sets, the macro
  - CLD_COLOURCONTROL_ATTR_NUMBER_OF_PRIMARIES

    is used to define the required number of colour primaries, N, in the range 1 to 6 (0xFF can also be specified if the number of primaries is not known). This macro is used to automatically enable the required attributes from these attribute sets - for example, if N is set to 4 then the following attributes are enabled:

    `u16Primary1X`, `u16Primary1Y`, `u8Primary1Intensity`, `u16Primary2X`, `u16Primary2Y`, `u8Primary2Intensity`, `u16Primary3X`, `u16Primary3Y`, `u8Primary3Intensity`, `u16Primary4X`, `u16Primary4Y`, `u8Primary4Intensity`.
- For optional attributes from the ‘Defined Colour Points Settings’ attribute set:
  - CLD_COLOURCONTROL_ATTR_WHITE_POINT_X
  - CLD_COLOURCONTROL_ATTR_WHITE_POINT_Y
  - CLD_COLOURCONTROL_ATTR_COLOUR_POINT_R_X
  - CLD_COLOURCONTROL_ATTR_COLOUR_POINT_R_Y
  - CLD_COLOURCONTROL_ATTR_COLOUR_POINT_R_INTENSITY
  - CLD_COLOURCONTROL_ATTR_COLOUR_POINT_G_X

  - CLD_COLOURCONTROL_ATTR_COLOUR_POINT_G_Y
  - CLD_COLOURCONTROL_ATTR_COLOUR_POINT_G_INTENSITY
  - CLD_COLOURCONTROL_ATTR_COLOUR_POINT_B_X
  - CLD_COLOURCONTROL_ATTR_COLOUR_POINT_B_Y
  - CLD_COLOURCONTROL_ATTR_COLOUR_POINT_B_INTENSITY
- For optional attributes from the 'Enhanced Colour Mode' attributes, the following must be defined:
  - CLD_COLOURCONTROL_ATTR_ENHANCED_COLOUR_MODE
  - CLD_COLOURCONTROL_ATTR_COLOUR_CAPABILITIES

    The required 'Enhanced Colour Mode' attributes for a device must then be enabled through a 'Colour Capabilities' Definition (see below).

Global Attributes

Add this line to enable the optional Attribute Reporting Status attribute:

```
#define CLD_COLOURCONTROL_ATTR_ID_ATTRIBUTE_REPORTING_STATUS
```

Add this line to define the value (n) of the Cluster Revision attribute:

```
#define CLD_COLOURCONTROL_CLUSTER_REVISION <n>
```

The default value is 1, which corresponds to the revision of the cluster in the ZCL r6 specification (see Section 2.4).

'Colour Capabilities' Definition

If required, certain 'Colour Information' attributes and all 'Enhanced Colour Mode' attributes must be enabled through a 'Colour Capabilities' definition. Attributes are enabled as a group according to the required capability/ functionality. The capabilities are detailed in the table below, with their corresponding attributes and macros.

Table 48. 'Colour Capabilities' Macros

Capability/ Functionality	Attributes	Macro
Hue/Saturation	`u8CurrentHue` `u8CurrentSaturation`	COLOUR_CAPABILITY_HUE_SATURATION_SUPPORTED
Enhanced Hue (also need Hue/ Saturation)	`u16EnhancedCurrentHue*`	COLOUR_CAPABILITY_ENHANCE_HUE_SUPPORTED

Colour Loop (also need Enhanced Hue)	`u8ColourLoopActive*`
	`u8ColourLoopDirection*`
	`u16ColourLoopTime*`
	`u16ColourLoopStartEnhancedHue*`
	`u16ColourLoopStoredEnhancedHue*`
		COLOUR_CAPABILITY_COLOUR_LOOP_SUPPORTED

CIE XY Values (this is mandatory)	`u16CurrentX` `u16CurrentY`	COLOUR_CAPABILITY_XY_SUPPORTED

Colour Temperature	`u16ColourTemperatureMired`
	`u16ColourTemperatureMiredPhyMin*`
	`u16ColourTemperatureMiredPhyMax*`
	`u16ColourTemperatureMiredMin*`
	`u16ColourTemperatureMiredMax*`
	`bColourCoupleTemperatureMired-ToLevel*`
		COLOUR_CAPABILITY_COLOUR_TEMPERATURE_SUPPORTED

Table 48. 'Colour Capabilities' Macros*...continued*

Capability/ Functionality	Attributes	Macro
	u16StartupColourTemperatureMired*	

* The 'Enhanced Colour Mode' attributes also require 'enhanced colour mode' to be enabled through `#define CLD_COLOURCONTROL_ATTR_ENHANCED_COLOUR_MODE`

The above macros will automatically invoke the macros for the individual attributes in the capability group, e.g. E_CLD_COLOURCONTROL_ATTR_CURRENT_HUE for the attribute `u8CurrentHue`.

The enabled Colour Capabilities are reflected in the 'Enhanced Colour Mode' attribute (bitmap) `u16ColourCapabilities`.

Example Colour Capabilities definitions are provided below for different devices.

ZLO Extended Colour Light:

```
#define CLD_COLOURCONTROL_COLOUR_CAPABILITIES
          (COLOUR_CAPABILITY_HUE_SATURATION_SUPPORTED | \\
         COLOUR_CAPABILITY_ENHANCE_HUE_SUPPORTED  | \\
         COLOUR_CAPABILITY_COLOUR_LOOP_SUPPORTED | \\
         COLOUR_CAPABILITY_XY_SUPPORTED | \\
         COLOUR_CAPABILITY_COLOUR_TEMPERATURE_SUPPORTED)
```

ZLO Colour Light:

```
#define CLD_COLOURCONTROL_COLOUR_CAPABILITIES
              (COLOUR_CAPABILITY_HUE_SATURATION_SUPPORTED | \\
              COLOUR_CAPABILITY_ENHANCE_HUE_SUPPORTED    | \\
              COLOUR_CAPABILITY_COLOUR_LOOP_SUPPORTED    | \\
              COLOUR_CAPABILITY_XY_SUPPORTED)
```

ZLO Colour Temperature Light:

```
#define CLD_COLOURCONTROL_COLOUR_CAPABILITIES
        (COLOUR_CAPABILITY_COLOUR_TEMPERATURE_SUPPORTED)
```

32 Ballast Configuration Cluster

This chapter describes the Ballast Configuration cluster, which is concerned with a configuring a lighting ballast that restricts the amount of light emitted by a set of lamps connected to the ballast.

The Ballast Configuration cluster has a Cluster ID of 0x0301.

32.1 Overview

The Ballast Configuration cluster allows the ballast for a set of lamps to be configured.

To use the functionality of this cluster, you must include the file **BallastConfiguration.h** in your application and enable the cluster by defining CLD_BALLAST_CONFIGURATION in the **zcl_options.h** file.

It is also necessary to enable the cluster as a server or client, or as both:

- The cluster server is able to receive commands to access ballast configuration data on the local device.
- The cluster client is able to send commands to access ballast configuration data on the remote device.

The inclusion of the client or server software must be pre-defined in the application's compile-time options (in addition, if the cluster is to reside on a custom endpoint then the role of client or server must also be specified when creating the cluster instance). The compile-time options for the Ballast Configuration cluster are fully detailed in Section 32.5.

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

- Ballast Information
- Ballast Settings
- Lamp Information
- Lamp Settings
- Global

Note that not all of the above attribute sets are currently implemented in the NXP software and not all attributes within a supported attribute set are implemented (see Section 32.2 for the supported attribute sets and attributes).

32.2 Cluster structure and attributes

The structure definition for the Device Temperature Configuration cluster is:

```
typedef struct
{
#ifdef BALLAST_CONFIGURATION_SERVER
/* Ballast Information attribute set */
#ifdef CLD_BALLASTCONFIGURATION_ATTR_PHYSICAL_MIN_LEVEL
    zuint8                  u8PhysicalMinLevel;
#endif
#ifdef CLD_BALLASTCONFIGURATION_ATTR_PHYSICAL_MAX_LEVEL
    zuint8                  u8PhysicalMaxLevel;
#endif
    zbmap8                  u8BallastStatus;
/* Ballast Settings attribute set */
#ifdef CLD_BALLASTCONFIGURATION_ATTR_MIN_LEVEL
    zuint8                  u8MinLevel;
#endif
#ifdef CLD_BALLASTCONFIGURATION_ATTR_MAX_LEVEL
    zuint8                  u8MaxLevel;
#endif
```

```
#ifdef CLD_BALLASTCONFIGURATION_ATTR_POWER_ON_LEVEL
    zuint8                  u8PowerOnLevel;
#endif
#ifdef CLD_BALLASTCONFIGURATION_ATTR_POWER_ON_FADE_TIME
    zuint16                 u16PowerOnFadeTime;
#endif
#ifdef CLD_BALLASTCONFIGURATION_ATTR_INTRINSIC_BALLAST_FACTOR
    zuint8                  u8IntrinsicBallastFactor;
#endif
#ifdef CLD_BALLASTCONFIGURATION_ATTR_BALLAST_FACTOR_ADJUSTMENT
    zuint8                  u8BallastFactorAdjustment;
#endif
/* Lamp Information attribute set */
#ifdef CLD_BALLASTCONFIGURATION_ATTR_LAMP_QUANTITY
    zuint8                  u8LampQuantity;
#endif
/* Lamp Settings attribute set */
#ifdef CLD_BALLASTCONFIGURATION_ATTR_LAMP_TYPE
    tsZCL_CharacterString   sLampType;
    uint8                   au8LampType[16];
#endif
#ifdef CLD_BALLASTCONFIGURATION_ATTR_LAMP_MANUFACTURER
    tsZCL_CharacterString   sLampManufacturer;
    uint8                   au8LampManufacturer[16];
#endif
#ifdef CLD_BALLASTCONFIGURATION_ATTR_LAMP_RATED_HOURS
    zuint24                 u32LampRatedHours;
#endif
#ifdef CLD_BALLASTCONFIGURATION_ATTR_LAMP_BURN_HOURS
    zuint24                 u32LampBurnHours;
#endif
#ifdef CLD_BALLASTCONFIGURATION_ATTR_LAMP_ALARM_MODE
    zbmap8                  u8LampAlarmMode;
#endif
#ifdef CLD_BALLASTCONFIGURATION_ATTR_LAMP_BURN_HOURS_TRIP_POINT
    zuint24                 u32LampBurnHoursTripPoint;
#endif
#endif
    zuint16                 u16ClusterRevision;
} tsCLD_BallastConfiguration;
```

In some attributes described below, a light level is specified as an 8-bit value. This is mapped to a percentage light level by means of a manufacturer-defined light curve, where 0x01 corresponds to 0.1% and 0xFE corresponds to 100% (0xFF is reserved).

Ballast Information Attribute Set

```
u8PhysicalMinLevel is an optional attribute representing the minimum light level that the
u8PhysicalMaxLevel is an optional attribute representing the maximum light level that the
u8BallastStatus is a mandatory attribute containing a bitmap which indicates the status of
```

Bits	Status
0	Ballast operational status: 0: Ballast is fully operational 1: Ballast is not fully operational
1	Lamp status:

Bits	Status
	0: All associated lamps are in their sockets 1: Not all associated lamps are in their sockets
2-7	Reserved

Ballast Settings Attribute Set

```
u8MinLevel is an optional attribute representing the minimum light level that the lamps ar
u8MaxLevel is an optional attribute representing the maximum light level that the lamps ar
u8PowerOnLevel is an optional attribute representing the light level that the will be prod
off) should be implemented on switch-on.
u16PowerOnFadeTime is an optional attribute representing the time, in tenths of a second,
on.
u8IntrinsicBallastFactor is an optional attribute representing the ballast factor of the b
lamp combination, as a percentage. This is a multiplication factor which, if used, is appl
u8BallastFactorAdjustment is an optional attribute representing a multiplication factor, a
```

Lamp Information Attribute Set

```
u8LampQuantity is an optional attribute indicating the number of lamps connected to the ba
```

Lamp Settings Attribute Set

- The following optional pair of attributes are used to store a human readable description of the type of lamp connected to the ballast:

```
sLampType is a tsZCL_CharacterString structure (see Section 6.1.14) for a string of up to
au8LampType[16] is a byte-
array which contains the character data bytes representing the lamp type
```

- The following optional pair of attributes are used to store a human readable name of the manufacturer of the lamps connected to the ballast:

```
sLampManufacturer is a tsZCL_CharacterString structure (see Section 6.1.14) for a string o
au8LampManufacturer[16] is a byte-
array which contains the character data bytes representing the manufacturer name
u32LampRatedHours is an optional 24-
bit attribute indicating the manufacturer's estimated lifetime of the lamps, in hours, in
u32LampBurnHours is an optional 24-
bit attribute indicating the cumulative total hours of operation of the lamps (only the ho
u8LampAlarmMode is an optional attribute containing a bitmap that specifies the attributes
```

Bits	Alarm Trigger
0	Alarm triggered when `u32LampBurnHours` reaches `u32LampBurnHoursTripPoint`: 0: Alarm trigger disabled 1: Alarm trigger enabled
1-7	Reserved

```
u32LampBurnHoursTripPoint is an optional attribute specifying the number of hours of opera
```

Global Attribute Set

```
u16ClusterRevision is a mandatory attribute that specifies the revision of the cluster spe
```

32.3 Functions

The following Ballast Configuration cluster function is provided in the NXP implementation of the ZCL:

Function	Page
eCLD_BallastConfigurationCreateBallastConfiguration	713

32.3.1 eCLD_BallastConfigurationCreateBallastConfiguration

```
teZCL_Status eCLD_BallastConfigurationCreateBallastConfiguration(
        tsZCL_ClusterInstance *psClusterInstance,
        bool_t bIsServer,
        tsZCL_ClusterDefinition *psClusterDefinition,
        void *pvEndPointSharedStructPtr,
        uint8 *pu8AttributeControlBits);
```

Description

This function creates an instance of the Ballast Configuration 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 Ballast Configuration cluster instance on the endpoint, but instances of other clusters may also be created on the same endpoint by calling their corresponding creation functions.

Note: *This function must not be called for an endpoint on which a standard ZigBee device is used. In this case, the device and its supported clusters must be registered on the endpoint using the relevant device registration function.*

When used, this function must be called after the stack has been started and after the ZCL has been initialized.

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 Ballast Configuration cluster.

The function initializes the array elements to zero.

Parameters

psClusterInstance Pointer to structure containing information about the cluster instance to be created (see Section 6.1.16). This structure is updated by the function by initialising individual structure fields.

bIsServer Type of cluster instance (server or client) to be created:

TRUE - server

FALSE - client

psClusterDefinition Pointer to structure indicating the type of cluster to be created (see Section 6.1.2). In this case, this structure must contain the details of the Ballast Configuration cluster. This parameter can refer to a pre-filled structure called `sCLD_BallastConfiguration` which is provided in the **BallastConfiguration.h** file.

pvEndPointSharedStructPtr Pointer to the shared structure used for attribute storage. This parameter should be the address of the structure of type `tsCLD_BallastConfigurationn` which defines the attributes of Ballast Configuration cluster. The function initializes the attributes with default values.
pu8AttributeControlBits Pointer to an array of **uint8** values, with one element for each attribute in the cluster (see above).

Returns

E_ZCL_SUCCESS
E_ZCL_ERR_PARAMETER_NULL

32.4 Enumerations

32.4.1 teCLD_BallastConfiguration_ClusterID

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

```
typedef enum
{
    /* Ballast Information attribute set attribute IDs */
    E_CLD_BALLASTCONFIGURATION_ATTR_PHYSICAL_MIN_LEVEL      = 0x0000,
    E_CLD_BALLASTCONFIGURATION_ATTR_PHYSICAL_MAX_LEVEL,
    E_CLD_BALLASTCONFIGURATION_ATTR_BALLAST_STATUS,
    /* Ballast Settings attribute set attribute IDs */
    E_CLD_BALLASTCONFIGURATION_ATTR_MIN_LEVEL               = 0x0010,
    E_CLD_BALLASTCONFIGURATION_ATTR_MAX_LEVEL,
    E_CLD_BALLASTCONFIGURATION_ATTR_POWER_ON_LEVEL,
    E_CLD_BALLASTCONFIGURATION_ATTR_POWER_ON_FADE_TIME,
    E_CLD_BALLASTCONFIGURATION_ATTR_INTRINSIC_BALLAST_FACTOR,
    E_CLD_BALLASTCONFIGURATION_ATTR_BALLAST_FACTOR_ADJUSTMENT,
    /* Lamp Information attribute set attribute IDs */
    E_CLD_BALLASTCONFIGURATION_ATTR_LAMP_QUANTITY           = 0x0020,
    /* Lamp Settings attribute set attribute IDs */
    E_CLD_BALLASTCONFIGURATION_ATTR_LAMP_TYPE               = 0x0030,
    E_CLD_BALLASTCONFIGURATION_ATTR_LAMP_MANUFACTURER,
    E_CLD_BALLASTCONFIGURATION_ATTR_LAMP_RATED_HOURS,
    E_CLD_BALLASTCONFIGURATION_ATTR_LAMP_BURN_HOURS,
    E_CLD_BALLASTCONFIGURATION_ATTR_LAMP_ALARM_MODE,
    E_CLD_BALLASTCONFIGURATION_ATTR_LAMP_BURN_HOURS_TRIP_POINT,
} teCLD_BallastConfiguration_ClusterID;
```

32.5 Compile-time options

To enable the Ballast Configuration cluster in the code to be built, it is necessary to add the following to the **zcl_options.h** file:

```
#define CLD_BALLASTCONFIGURATION
```

In addition, to include the software for a cluster client or server or both, it is necessary to add one or both of the following to the same file:

```
#define BALLASTCONFIGURATION_CLIENT
#define BALLASTCONFIGURATION_SERVER
```

The Ballast Configuration cluster contains macros that may be optionally specified at compile-time by adding some or all the following lines to the **zcl_options.h** file.

Optional Attributes

Add this line to enable the optional Physical Minimum Level attribute:

```
#define E_CLD_BALLASTCONFIGURATION_ATTR_PHYSICAL_MIN_LEVEL
```

Add this line to enable the optional Physical Maximum Level attribute:

```
#define E_CLD_BALLASTCONFIGURATION_ATTR_PHYSICAL_MAX_LEVEL
```

Add this line to enable the optional Ballast Status attribute:

```
#define E_CLD_BALLASTCONFIGURATION_ATTR_BALLAST_STATUS
```

Add this line to enable the optional Minimum Level attribute:

```
#define E_CLD_BALLASTCONFIGURATION_ATTR_MIN_LEVEL
```

Add this line to enable the optional Maximum Level attribute:

```
#define E_CLD_BALLASTCONFIGURATION_ATTR_MAX_LEVEL
```

Add this line to enable the optional Power-on Level attribute:

```
#define E_CLD_BALLASTCONFIGURATION_ATTR_POWER_ON_LEVEL
```

Add this line to enable the optional Power-on Fade Time attribute:

```
#define E_CLD_BALLASTCONFIGURATION_ATTR_POWER_ON_FADE_TIME
```

Add this line to enable the optional Intrinsic Ballast Factor attribute:

```
#define E_CLD_BALLASTCONFIGURATION_ATTR_INTRINSIC_BALLAST_FACTOR
```

Add this line to enable the optional Ballast Factor Adjustment attribute:

```
#define E_CLD_BALLASTCONFIGURATION_ATTR_BALLAST_FACTOR_ADJUSTMENT
```

Add this line to enable the optional Lamp Quantity attribute:

```
#define E_CLD_BALLASTCONFIGURATION_ATTR_LAMP_QUANTITY
```

Add this line to enable the optional Lamp Type attributes:

```
#define E_CLD_BALLASTCONFIGURATION_ATTR_LAMP_TYPE
```

Add this line to enable the optional Lamp Manufacturer attributes:

```
#define E_CLD_BALLASTCONFIGURATION_ATTR_LAMP_MANUFACTURER
```

Add this line to enable the optional Lamp Rated Hours attribute:

```
#define E_CLD_BALLASTCONFIGURATION_ATTR_LAMP_MANUFACTURER
```

Add this line to enable the optional Lamp Burn Hours attribute:

```
#define E_CLD_BALLASTCONFIGURATION_ATTR_LAMP_BURN_HOURS
```

Add this line to enable the optional Lamp Alarm Mode attribute:

```
#define E_CLD_BALLASTCONFIGURATION_ATTR_LAMP_ALARM_MODE
```

Add this line to enable the optional Lamp Burn Hours Trip Point attribute:

```
#define E_CLD_BALLASTCONFIGURATION_ATTR_LAMP_BURN_HOURS_TRIP_POINT
```

Global Attributes

Add this line to define the value (n) of the Cluster Revision attribute:

```
#define CLD_BALLASTCONFIGURATION_CLUSTER_REVISION <n>
```

The default value is 1, which corresponds to the revision of the cluster in the ZCL r6 specification (see Section 2.4).

Part VI: HVAC Clusters

This part comprises three chapters:

- Chapter 33 details the **Thermostat** cluster
- Chapter 34 details the **Fan Control** cluster
- Chapter 35 details the **Thermostat UI Configuration** cluster

33 Thermostat Cluster

This chapter outlines the Thermostat cluster, which provides an interface for configuring and controlling the functionality of a thermostat.

The Thermostat cluster has a Cluster ID of 0x0201.

33.1 Overview

The Thermostat cluster is required in ZigBee devices as indicated in the table below.

Table 49. Thermostat Cluster in ZigBee Devices

	Server-side	Client-side
Mandatory in...	Thermostat	
Optional in...		Remote Control

The Thermostat cluster is enabled by defining CLD_THERMOSTAT in the **zcl_options.h** file.

The inclusion of the client or server software must be pre-defined in the application's compile-time options (in addition, if the cluster is to reside on a custom endpoint then the role of client or server must also be specified when creating the cluster instance).

The compile-time options for the Thermostat cluster are fully detailed in Section 33.10.

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

- Thermostat Information
- Thermostat Settings

The attributes are listed and described next, in Section 33.2.

33.2 Thermostat Cluster structure and attributes

The Thermostat cluster is contained in the following `tsCLD_Thermostat` structure:

```
typedef struct
{
#ifdef THERMOSTAT_SERVER
    zint16                  i16LocalTemperature;
#ifdef CLD_THERMOSTAT_ATTR_OUTDOOR_TEMPERATURE
    zint16                  i16OutdoorTemperature;
#endif
#ifdef CLD_THERMOSTAT_ATTR_OCCUPANCY
    zbmap8                  u8Occupancy;
#endif
#ifdef CLD_THERMOSTAT_ATTR_ABS_MIN_HEAT_SETPOINT_LIMIT
    zint16                  i16AbsMinHeatSetpointLimit;
#endif
#ifdef CLD_THERMOSTAT_ATTR_ABS_MAX_HEAT_SETPOINT_LIMIT
    zint16                  i16AbsMaxHeatSetpointLimit;
#endif
#ifdef CLD_THERMOSTAT_ATTR_ABS_MIN_COOL_SETPOINT_LIMIT
    zint16                  i16AbsMinCoolSetpointLimit;
#endif
#ifdef CLD_THERMOSTAT_ATTR_ABS_MAX_COOL_SETPOINT_LIMIT
    zint16                  i16AbsMaxCoolSetpointLimit;
#endif
```

```
#ifdef CLD_THERMOSTAT_ATTR_PI_COOLING_DEMAND
    zuint8                  u8PICoolingDemand;
#endif
#ifdef CLD_THERMOSTAT_ATTR_PI_HEATING_DEMAND
    zuint8                  u8PIHeatingDemand;
#endif
/* Thermostat settings attribute set attribute IDs */
#ifdef CLD_THERMOSTAT_ATTR_LOCAL_TEMPERATURE_CALIBRATION
    zint8                   i8LocalTemperatureCalibration;
#endif
    zint16                  i16OccupiedCoolingSetpoint;
    zint16                  i16OccupiedHeatingSetpoint;
#ifdef CLD_THERMOSTAT_ATTR_UNOCCUPIED_COOLING_SETPOINT
    zint16                  i16UnoccupiedCoolingSetpoint;
#endif
#ifdef CLD_THERMOSTAT_ATTR_UNOCCUPIED_HEATING_SETPOINT
    zint16                  i16UnoccupiedHeatingSetpoint;
#endif
#ifdef CLD_THERMOSTAT_ATTR_MIN_HEAT_SETPOINT_LIMIT
    zint16                  i16MinHeatSetpointLimit;
#endif
#ifdef CLD_THERMOSTAT_ATTR_MAX_HEAT_SETPOINT_LIMIT
    zint16                  i16MaxHeatSetpointLimit;
#endif
#ifdef CLD_THERMOSTAT_ATTR_MIN_COOL_SETPOINT_LIMIT
    zint16                  i16MinCoolSetpointLimit;
#endif
#ifdef CLD_THERMOSTAT_ATTR_MAX_COOL_SETPOINT_LIMIT
    zint16                  i16MaxCoolSetpointLimit;
#endif
#ifdef CLD_THERMOSTAT_ATTR_MIN_SETPOINT_DEAD_BAND
    zint8                   i8MinSetpointDeadBand;
#endif
#ifdef CLD_THERMOSTAT_ATTR_REMOTE_SENSING
    zbmap8                  u8RemoteSensing;
#endif
    zenum8                  eControlSequenceOfOperation;
    zenum8                  eSystemMode;
#ifdef CLD_THERMOSTAT_ATTR_ALARM_MASK
    zbmap8                  u8AlarmMask;
#endif
#ifdef CLD_THERMOSTAT_ATTR_ATTRIBUTE_REPORTING_STATUS
    zenum8                  u8AttributeReportingStatus;
#endif
#endif
    zuint16                 u16ClusterRevision;
} tsCLD_Thermostat;
```

where:

‘Thermostat Information’ Attribute Set

- `i16LocalTemperature` is a mandatory attribute representing the measured temperature in degrees Celsius, as follows:

  `i16LocalTemperature` = 100 x temperature in degrees Celsius

  The possible values are used as follows:

  0x0000 to 0x7FFF represent positive temperatures from 0°C to 327.67ºC

  0x8000 indicates that the temperature measurement is invalid

0x8001 to 0x954C are unused values
0x954D to 0xFFFF represent negative temperatures from -273.15°C to -1°C (in two's complement form)

- `i16OutdoorTemperature` is an optional attribute representing the outside temperature in degrees Celsius. This temperature is represented as described above for `i16LocalTemperature`.
- `u8Occupancy` is an optional attribute indicating whether the heated/cooled space has been detected as occupied. Bit 0 is used as a flag as follows (all other bits are reserved):
  - 1 = occupied
  - 0 = not occupied
- `i16AbsMinHeatSetpointLimit` is an optional attribute specifying the absolute minimum possible temperature of the heating setpoint (as determined by the manufacturer). This temperature is represented as described above for `i16LocalTemperature`.
- `i16AbsMaxHeatSetpointLimit` is an optional attribute specifying the absolute maximum possible temperature of the heating setpoint (as determined by the manufacturer). This temperature is represented as described above for `i16LocalTemperature`.
- `i16AbsMinCoolSetpointLimit` is an optional attribute specifying the absolute minimum possible temperature of the cooling setpoint (as determined by the manufacturer). This temperature is represented as described above for `i16LocalTemperature`.
- `i16AbsMaxCoolSetpointLimit` is an optional attribute specifying the absolute maximum possible temperature of the cooling setpoint (as determined by the manufacturer). This temperature is represented as described above for `i16LocalTemperature`.

'Thermostat Settings' Attribute Set

- `u8PICoolingDemand` is an optional attribute indicating the level of cooling required by the PI (Proportional Integral) control loop, if any, used by the thermostat. It is a percentage value and takes the value 0 when the thermostat is 'off' or in 'heating' mode.
- `u8PIHeatingDemand` is an optional attribute indicating the level of heating required by the PI (Proportional Integral) control loop, if any, used by the thermostat. It is a percentage value and takes the value 0 when the thermostat is 'off' or in 'cooling' mode.
- `i8LocalTemperatureCalibration` is an optional attribute representing a temperature offset (in the range -2.5°C to 2.5°C) that can be added to or subtracted from the displayed temperature:
  `i8LocalTemperatureCalibration` = 100 x offset in degrees Celsius
  The possible values are used as follows:
  0x00 to 0x19 represent positive offsets from 0°C to 2.5ºC
  0x20 to 0xE6 are unused values
  0xE7 to 0xFF represent negative offets from -2.5°C to -1°C (in two's complement form)
- `i16OccupiedCoolingSetpoint` is an optional attribute specifying the cooling setpoint (target temperature) when the cooling space is occupied. The value is calculated as described above for the `i16LocalTemperature` attribute and must take a value in the range defined by the attributes `i16MinCoolSetpointLimit` and `i16MaxCoolSetpointLimit`. If it is not known whether the space is occupied, this attribute will be used as the cooling setpoint (rather than `i16UnoccupiedCoolingSetpoint`).
- `i16OccupiedHeatingSetpoint` is an optional attribute specifying the heating setpoint (target temperature) when the heating space is occupied. The value is calculated as described above for the `i16LocalTemperature` attribute and must take a value in the range defined by the attributes `i16MinHeatSetpointLimit` and `i16MaxHeatSetpointLimit`. If it is not known whether the space is occupied, this attribute will be used as the heating setpoint (rather than `i16UnoccupiedHeatingSetpoint`).

Note: *`i16OccupiedCoolingSetpoint` must always be greater in value than `i16OccupiedHeatingSetpoint` by an amount at least equal to the value of `i8MinSetpointDeadBand` (below). An attempt to violate this condition will result in a default response with the status INVALID_VALUE.*

- `i16UnoccupiedCoolingSetpoint` is an optional attribute specifying the cooling setpoint (target temperature) when the cooling space is unoccupied. The value is calculated as described above for the `i16LocalTemperature` attribute and must take a value in the range defined by the attributes `i16AbsMinCoolSetpointLimit` and `i16MaxCoolSetpointLimit`. If it is not known whether the space is occupied, this attribute will not be used (`i16OccupiedCoolingSetpoint` will be used instead).
- `i16UnoccupiedHeatingSetpoint` is an optional attribute specifying the heating setpoint (target temperature) when the heating space is unoccupied. The value is calculated as described above for the `i16LocalTemperature` attribute and must take a value in the range defined by the attributes `i16MinHeatSetpointLimit` and `i16MaxHeatSetpointLimit`. If it is not known whether the space is occupied, this attribute will not be used (`i16OccupiedHeatingSetpoint` will be used instead).

Note: *`i16UnoccupiedCoolingSetpoint` must always be greater in value than `i16UnoccupiedHeatingSetpoint` by an amount at least equal to the value of `i8MinSetpointDeadBand` (below). An attempt to violate this condition will result in a default response with the status INVALID_VALUE.*

- `i16MinHeatSetpointLimit` is an optional attribute specifying the minimum possible temperature of the heating setpoint. This temperature is represented as described above for `i16LocalTemperature`. The value set must be greater than or equal to the value of `i16AbsMinHeatSetpointLimit`, which is also the default value for this attribute.
- `i16MaxHeatSetpointLimit` is an optional attribute specifying the maximum possible temperature of the heating setpoint. This temperature is represented as described above for `i16LocalTemperature`. The value set must be less than or equal to the value of `i16AbsMaxHeatSetpointLimit`, which is also the default value for this attribute.
- `i16MinCoolSetpointLimit` is an optional attribute specifying the minimum possible temperature of the cooling setpoint. This temperature is represented as described above for `i16LocalTemperature`. The value set must be greater than or equal to the value of `i16AbsMinCoolSetpointLimit`, which is also the default value for this attribute.
- `i16MaxCoolSetpointLimit` is an optional attribute specifying the maximum possible temperature of the cooling setpoint. This temperature is represented as described above for `i16LocalTemperature`. The value set must be less than or equal to the value of `i16AbsMaxCoolSetpointLimit`, which is also the default value for this attribute.

Note: *The above four 'Limit' attributes can be set in the compile-time options using macros, as described in Section 33.10.*

- `i8MinSetpointDeadBand` is an optional attribute specifying the minimum difference between the heating setpoint and cooling setpoint, in steps of 0.1°C. The attribute can take a value in the range 0x0A to 0x19, representing 1°C to 2.5°C. All other values are unused.
- `u8RemoteSensing` is an optional attribute comprising an 8-bit bitmap which indicates whether remote (networked) or internal sensors are being used to measure/detect the local temperature, outside temperature and occupancy. The bitmap is detailed in the table below.

Bit	Description
0	**Local temperature** 1 - Remote sensor 0 - Internal sensor
1	**Outside temperature** 1 - Remote sensor 0 - Internal sensor

Bit	Description
2	**Occupancy** 1 - Remote sensor 0 - Internal sensor
3-7	Reserved

- `eControlSequenceOfOperation` is an optional attribute representing the operational capabilities/ environment of the thermostat. The possible values are indicated in the table below:

Value	Capabilities	Notes (see `eSystemMode`)
0x00	Cooling only	Heat and Emergency Heating are not possible
0x01	Cooling with Reheat	Heat and Emergency Heating are not possible
0x02	Heating only	Cool and Pre-cooling are not possible
0x03	Heating with Reheat	Cool and Pre-cooling are not possible
0x04	Cooling and Heating 4-pipes	All modes are possible
0x05	Cooling and Heating 4-pipes with Reheat	All modes are possible
0x06 – 0xFE	Reserved	-

- `eSystemMode` is an optional attribute specifying the current operating mode of the thermostat. The possible modes/values are indicated in the table below:

Value	Description
0x00	Off
0x01	Auto
0x02	Reserved
0x03	Cool
0x04	Heat
0x05	Emergency Heating
0x06	Pre-cooling
0x07	Fan only
0x08 – 0xFE	Reserved

- `u8AlarmMask` is an optional attribute containing a 3-bit bitmap specifying which alarms are enabled from those listed in the table below (use of the Alarms cluster is also required):

Bit	Description
0	**Initialisation failure** (device failed to complete initialization at power-up) 1 - Alarm enabled 0 - Alarm disabled
1	**Hardware failure** 1 - Alarm enabled 0 - Alarm disabled

Bit	Description
2	**Self-calibration failure** 1 - Alarm enabled 0 - Alarm disabled
3-7	Reserved

Global Attributes

- `u8AttributeReportingStatus` is an optional attribute that should be enabled when attribute reporting is used for the cluster (see Section 2.3.5). The value of this attribute indicates whether there are attribute reports still pending (0x00) or the attribute reports are complete (0x01) - all other values are reserved. This attribute is also described in Section 2.4.
- `u16ClusterRevision` is a mandatory attribute that specifies the revision of the cluster specification on which this cluster instance is based. The cluster specification in the ZCL r6 corresponds to a cluster revision of 1. The value is incremented by one for each subsequent revision of the cluster specification. This attribute is also described in Section 2.4.

33.3 Attributes for Default Reporting

The following attributes of the Thermostat cluster can be selected for default reporting:

```
i16LocalTemperature
u8PICoolingDemand
```

- `u8PIHeatingDemand`

Attribute reporting (including default reporting) is described in Appendix B. Enabling reports for these attributes is described in Appendix B.3.6.

33.4 Thermostat Operations

The Thermostat cluster server is mandatory for some HVAC devices, such as the Thermostat device, while the cluster client can be used on a controlling device, such as the Remote Control device.

The sections below describe common operations using the Thermostat cluster.

33.4.1 Initialisation

The function **eCLD_ThermostatCreateThermostat()** is used to create an instance of the Thermostat cluster. The function is generally called by the initialization function for the host device.

33.4.2 Recording and Reporting the Local Temperature

A record of the local temperature is kept in the mandatory attribute `i16LocalTemperature` on the cluster server - this attribute is fully detailed in Section 33.2. The value of this attribute can be updated by the server application using the function **eCLD_ThermostatSetAttribute()** - for example, as the result of a local temperature measurement.

The value of the attribute `i16LocalTemperature` can be regularly reported to a cluster client - for example, to allow the local temperature to be displayed to the user. This automated reporting can be configured and started on the server using the function **eCLD_ThermostatStartReportingLocalTemperature()**. Reports is sent regularly, but not periodically - maximum and minimum time-intervals between consecutive reports can be specified.

33.4.3 Configuring Heating and Cooling Setpoints

Functions are provided to update the following two optional attributes that are used to specify setpoints (target temperatures) for heating and cooling:

```
i16OccupiedHeatingSetpoint
i16OccupiedCoolingSetpoint
```

If both of these setpoints are used, the cooling setpoint value must be greater than the heating setpoint value. These attributes are fully detailed in Section 33.2.

These server attributes can be controlled remotely from a client using the function **eCLD_ThermostatCommandSetpointRaiseOrLowerSend()**, usually as the result of user input on a controlling device. This function is used on the client to send a SetpointRaiseOrLower command to the server to increase or decrease the value of one or both of these setpoint attributes by a specified amount. On receipt of this command, an E_CLD_THERMOSTAT_CMD_SETPOINT_RAISE_LOWER event is generated on the server to notify the server application.

The server application can modify the values of these attributes using the function **eCLD_ThermostatSetAttribute()**.

Note: *These and other attributes of the Thermostat cluster can also be written and read using the general attribute access functions, as described in Section 2.3.*

33.5 Thermostat Events

The Thermostat cluster has its own events that are handled through the callback mechanism outlined in Chapter 3. If a device uses the Thermostat cluster then Thermostat 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 **eHA_RegisterThermostatEndPoint()** for a Thermostat device). The relevant callback function will then be invoked when a Thermostat event occurs.

For a Thermostat 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 `tsCLD_ThermostatCallBackMessage` structure:

```
typedef struct
{
    uint8                       u8CommandId;
    union
    {
      tsCLD_Thermostat_SetpointRaiseOrLowerPayload
                               *psSetpointRaiseOrLowerPayload;
    } uMessage;
} tsCLD_ThermostatCallBackMessage;
```

The `u8CommandId` field of the above structure specifies the type of command that has been received - only one command type is possible and is described below.

E_CLD_THERMOSTAT_CMD_SETPOINT_RAISE_LOWER

In the `tsCLD_ThermostatCallBackMessage` structure, the `u8CommandId` is set to E_CLD_THERMOSTAT_CMD_SETPOINT_RAISE_LOWER on the Thermostat cluster server when a SetpointRaiseOrLower command has been received. On receipt of this command, the Thermostat command handler will be invoked.

33.6 Functions

The following Thermostat cluster functions are provided:

33.6.1 eCLD_ThermostatCreateThermostat

```
teZCL_Status eCLD_ThermostatCreateThermostat(
        tsZCL_ClusterInstance *psClusterInstance,
        bool_t bIsServer,
        sZCL_ClusterDefinition *psClusterDefinition,
        void *pvEndPointSharedStructPtr,
        uint8 *pu8AttributeControlBits,
        tsCLD_ThermostatCustomDataStructure
 psCustomDataStructure);
```

Description

This function creates an instance of the Thermostat 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 Thermostat cluster instance on the endpoint, but instances of other clusters may also be created on the same endpoint by calling their corresponding creation functions.

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

When used, this function must be the first Thermostat cluster function called in the application, and must be called after the stack has been started and after the ZCL has been initialized.

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 Thermostat cluster.

The function initializes the array elements to zero.

Parameters

psClusterInstance Pointer to structure containing information about the cluster instance to be created (see Section 6.1.16). This structure is updated by the function by initialising individual structure fields.

bIsServer Type of cluster instance (server or client) to be created:

TRUE - server

FALSE - client

psClusterDefinition Pointer to structure indicating the type of cluster to be created (see Section 6.1.2). In this case, this structure must contain the details of the Thermostat cluster. This parameter can refer to a pre-filled structure called `sCLD_Thermostat` which is provided in the **Thermostat.h** file.

pvEndPointSharedStructPtr Pointer to the shared structure used for attribute storage. This parameter should be the address of the structure of type `tsCLD_Thermostat` which defines the attributes of Thermostat cluster. The function initializes the attributes with default values.
pu8AttributeControlBits Pointer to an array of **uint8** values, with one element for each attribute in the cluster (see above).

Returns

E_ZCL_SUCCESS
E_ZCL_FAIL
E_ZCL_ERR_PARAMETER_NULL
E_ZCL_ERR_INVALID_VALUE

33.6.2 eCLD_ThermostatSetAttribute

```
teZCL_Status eCLD_ThermostatSetAttribute(
        uint8 u8SourceEndPointId,
        uint8 u8AttributeId,
        int16 i16AttributeValue);
```

Description

This function can be used on a Thermostat cluster server to update the Thermostat attributes - specifically to write a value to one of the following attributes:

```
i16LocalTemperature
i16OccupiedCoolingSetpoint
i16OccupiedHeatingSetpoint
```

The function first checks whether the value to be written falls within the valid range for the relevant attribute. If not, it returns with status E_ZCL_ERR_INVALID_VALUE. If the server attempts to write to an attribute other than those specified above, the function returns with status E_ZCL_DENY_ATTRIBUTE_ACCESS. If the cluster does not exist, it returns with status E_ZCL_ERR_CLUSTER_NOT_FOUND.

Parameters

u8SourceEndPointId Number of the endpoint on which the Thermostat cluster resides
u8AttributeId Identifier of attribute to be updated, one of:
E_CLD_THERMOSTAT_ATTR_ID_LOCAL_TEMPERATURE
E_CLD_THERMOSTAT_ATTR_ID_OCCUPIED_COOLING_SETPOINT
E_CLD_THERMOSTAT_ATTR_ID_OCCUPIED_HEATING_SETPOINT
i16AttributeValue Value to be written to attribute

Returns

E_ZCL_SUCCESS
E_ZCL_ERR_INVALID_VALUE
E_ZCL_DENY_ATTRIBUTE_ACCESS
E_ZCL_ERR_CLUSTER_NOT_FOUND

33.6.3 eCLD_ThermostatStartReportingLocalTemperature

```
teZCL_Status eCLD_ThermostatStartReportingLocalTemperature(
        uint8 u8SourceEndPointId,
        uint8 u8DstEndPointId,
        uint64 u64DstAddr,
        uint16 u16MinReportInterval,
        uint16 u16MaxReportInterval,
        int16 i16ReportableChange);
```

Description

This function can be used on a Thermostat cluster server to start automatic reporting of the measured local temperature to a cluster client. The change to be reported can be configured through this function. Reports is sent regularly (but not periodically), within the specified maximum and minimum time-intervals between consecutive reports.

Parameters

u8SourceEndPointId	Number of the local endpoint on which the Thermostat cluster server resides
u8DstEndPointId	Number of the endpoint to which reports are to be sent on the destination node
u64DstAddr	IEEE/MAC address of destination node
u16MinReportInterval	Minimum time-interval, in seconds, between reports
u16MaxReportInterval	Maximum time-interval, in seconds, between reports
i16ReportableChange	Specifies the change to be reported

Returns

E_ZCL_SUCCESS
E_ZCL_FAIL
E_ZCL_ERR_CLUSTER_NOT_FOUND

33.6.4 eCLD_ThermostatCommandSetpointRaiseOrLowerSend

```
teZCL_Status eCLD_ThermostatCommandSetpointRaiseOrLowerSend(
        uint8 u8SourceEndPointId,
        uint8 u8DestinationEndPointId,
        tsZCL_Address *psDestinationAddress,
        uint8 *pu8TransactionSequenceNumber,
        tsCLD_Thermostat_SetpointRaiseOrLowerPayload
 *psPayload);
```

Description

This function can be used on a Thermostat cluster client to send a ‘Setpoint Raise Or Lower’ command to the cluster server. This command is used to increase or decrease the heating setpoint and/or cooling setpoint by requesting a change to the values of the attribute `i16OccupiedHeatingSetpoint` and/or the attribute `i16OccupiedCoolingSetpoint`. The relevant setpoint(s) and the required temperature change are specified in the command payload structure t`sCLD_Thermostat_SetpointRaiseOrLowerPayload` (see Section 33.9.3).

A pointer must be specified to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is 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 is sent
u8DestinationEndPointId Number of the remote endpoint to which the request is sent
psDestinationAddress Pointer to a structure containing the address of the remote node to which the request is sent
pu8TransactionSequenceNumber Pointer to a location to store the Transaction Sequence Number (TSN) of the request
psPayload Pointer to the command payload (see Section 33.9.3)

Returns

E_ZCL_SUCCESS
E_ZCL_ERR_PARAMETER_NULL
E_ZCL_ERR_EP_RANGE
E_ZCL_ERR_EP_UNKNOWN
E_ZCL_ERR_CLUSTER_NOT_FOUND
E_ZCL_ERR_ZBUFFER_FAIL
E_ZCL_ERR_ZTRANSMIT_FAIL

33.7 Return codes

The Thermostat cluster functions use the ZCL return codes defined in Section 7.2.

33.8 Enumerations

33.8.1 'Attribute ID' Enumerations

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

```
typedef enum
{
    E_CLD_THERMOSTAT_ATTR_ID_LOCAL_TEMPERATURE  = 0x0000,
    E_CLD_THERMOSTAT_ATTR_ID_OUTDOOR_TEMPERATURE,
    E_CLD_THERMOSTAT_ATTR_ID_OCCUPANCY,
    E_CLD_THERMOSTAT_ATTR_ID_ABS_MIN_HEAT_SETPOINT_LIMIT,
    E_CLD_THERMOSTAT_ATTR_ID_ABS_MAX_HEAT_SETPOINT_LIMIT,
    E_CLD_THERMOSTAT_ATTR_ID_ABS_MIN_COOL_SETPOINT_LIMIT,
    E_CLD_THERMOSTAT_ATTR_ID_ABS_MAX_COOL_SETPOINT_LIMIT,
    E_CLD_THERMOSTAT_ATTR_ID_PI_COOLING_DEMAND,
    E_CLD_THERMOSTAT_ATTR_ID_PI_HEATING_DEMAND,
    E_CLD_THERMOSTAT_ATTR_ID_LOCAL_TEMPERATURE_CALIBRATION  = 0x0010,
    E_CLD_THERMOSTAT_ATTR_ID_OCCUPIED_COOLING_SETPOINT,
    E_CLD_THERMOSTAT_ATTR_ID_OCCUPIED_HEATING_SETPOINT,
    E_CLD_THERMOSTAT_ATTR_ID_UNOCCUPIED_COOLING_SETPOINT,
    E_CLD_THERMOSTAT_ATTR_ID_UNOCCUPIED_HEATING_SETPOINT,
    E_CLD_THERMOSTAT_ATTR_ID_MIN_HEAT_SETPOINT_LIMIT,
    E_CLD_THERMOSTAT_ATTR_ID_MAX_HEAT_SETPOINT_LIMIT,
    E_CLD_THERMOSTAT_ATTR_ID_MIN_COOL_SETPOINT_LIMIT,
    E_CLD_THERMOSTAT_ATTR_ID_MAX_COOL_SETPOINT_LIMIT,
```

```
    E_CLD_THERMOSTAT_ATTR_ID_MIN_SETPOINT_DEAD_BAND,
    E_CLD_THERMOSTAT_ATTR_ID_REMOTE_SENSING,
    E_CLD_THERMOSTAT_ATTR_ID_CONTROL_SEQUENCE_OF_OPERATION,
    E_CLD_THERMOSTAT_ATTR_ID_SYSTEM_MODE,
    E_CLD_THERMOSTAT_ATTR_ID_ALARM_MASK
} teCLD_Thermostat_AttributeID;
```

33.8.2 'Operating Capabilities' Enumerations

The following enumerations are used to set the optional attribute `eControlSequenceOfOperation` in the Thermostat cluster structure `tsCLD_Thermostat`.

```
typedef enum
{
    E_CLD_THERMOSTAT_CSOO_COOLING_ONLY = 0x00,
    E_CLD_THERMOSTAT_CSOO_COOLING_WITH_REHEAT,
    E_CLD_THERMOSTAT_CSOO_HEATING_ONLY,
    E_CLD_THERMOSTAT_CSOO_HEATING_WITH_REHEAT,
    E_CLD_THERMOSTAT_CSOO_COOLING_AND_HEATING_4_PIPES,
    E_CLD_THERMOSTAT_CSOO_COOLING_AND_HEATING_4_PIPES_WITH_REHEAT,
}teCLD_Thermostat_ControlSequenceOfOperation;
```

The above enumerations are described in the table below.

Table 50. 'Operating Capabilities' Enumerations

Enumeration	Description
E_CLD_THERMOSTAT_CSOO_COOLING_ONLY	Heat and Emergency Heating are not possible
E_CLD_THERMOSTAT_CSOO_COOLING_WITH_REHEAT	Heat and Emergency Heating are not possible
E_CLD_THERMOSTAT_CSOO_HEATING_ONLY	Cool and Pre-cooling are not possible
E_CLD_THERMOSTAT_CSOO_HEATING_WITH_REHEAT	Cool and Pre-cooling are not possible
E_CLD_THERMOSTAT_CSOO_COOLING_AND_HEATING_4_PIPES	All modes are possible
E_CLD_THERMOSTAT_CSOO_COOLING_AND_HEATING_4_PIPES_WITH_REHEAT	All modes are possible

33.8.3 'Command ID' Enumerations

The following enumeration is used to specify the type of command sent to a Thermostat cluster server.

```
typedef enum
{
   E_CLD_THERMOSTAT_CMD_SETPOINT_RAISE_LOWER = 0x00,
} teCLD_Thermostat_Command;
```

The above enumerations are described in the table below.

Table 51. 'Command ID' Enumerations

Enumeration	Command
E_CLD_THERMOSTAT_CMD_SETPOINT_RAISE_LOWER	Setpoint Raise Or Lower

33.8.4 'Setpoint Raise Or Lower' Enumerations

The following enumerations are used to specify an operating mode (heating, cooling or both) or the Thermostat.

```
{
    E_CLD_THERMOSTAT_SRLM_HEAT = 0x00,
    E_CLD_THERMOSTAT_SRLM_COOL,
    E_CLD_THERMOSTAT_SRLM_BOTH
}teCLD_Thermostat_SetpointRaiseOrLowerMode;
```

The above enumerations are described in the table below.

Table 52. 'Setpoint Raise Or Lower' Enumerations

Enumeration	Description
E_CLD_THERMOSTAT_SRLM_HEAT	Heating mode
E_CLD_THERMOSTAT_SRLM_COOL	Cooling mode
E_CLD_THERMOSTAT_SRLM_BOTH	Heating and Cooling modes

33.9 Structures

33.9.1 Custom Data Structure

The Thermostat cluster requires extra storage space to be allocated for use by internal functions. The structure definition for this storage is shown below:

```
typedef struct
{
    tsZCL_ReceiveEventAddress         sReceiveEventAddress;
    tsZCL_CallBackEvent               sCustomCallBackEvent;
    tsCLD_ThermostatCallBackMessage   sCallBackMessage;
} tsCLD_ThermostatCustomDataStructure;
```

The fields are for internal use and no knowledge of them is required.

33.9.2 tsCLD_ThermostatCallBackMessage

For a Thermostat cluster 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 `tsCLD_ThermostatCallBackMessage` structure:

```
typedef struct
{
    uint8                                           u8CommandId;
    union
    {
      tsCLD_Thermostat_SetpointRaiseOrLowerPayload *psSetpointRaiseOrLowerPayload;
    } uMessage;
} tsCLD_ThermostatCallBackMessage;
```

where:

- `u8CommandId` indicates the type of Thermostat cluster command that has been received - there is only one possibility: E_CLD_THERMOSTAT_CMD_SETPOINT_RAISE_LOWER

- `uMessage` is a union containing the command payload in the following form: `psSetpointRaiseOrLowerPayload` is a pointer to a structure containing the payload of a ‘Setpoint Raise Or Lower’ command - see Section 33.9.3.

33.9.3 tsCLD_Thermostat_SetpointRaiseOrLowerPayload

This stucture contains the payload of a ‘Setpoint Raise Or Lower’ command (from the cluster client) which requests a change the value of the attribute `i16OccupiedHeatingSetpoint` and/or the attribute `i16OccupiedCoolingSetpoint`.

```
typedef struct
{
    zenum8                  eMode;
    zint8                   i8Amount;
}tsCLD_Thermostat_SetpointRaiseOrLowerPayload;
```

where:

- `eMode` indicates the Thermostat operating mode to which the command relates, one of:
  - E_CLD_THERMOSTAT_SRLM_HEAT (Heating)
  - E_CLD_THERMOSTAT_SRLM_COOL (Cooling)
  - E_CLD_THERMOSTAT_SRLM_BOTH (Heating and Cooling)
- `i8Amount` represents the value (in two’s complement form) by which the setpoint corresponding to the specified operating mode is to be changed

33.10 Compile-time options

To enable the Thermostat cluster in the code to be built, it is necessary to add the following to the **zcl_options.h** file:

```
#define CLD_THERMOSTAT
```

In addition, to include the software for a cluster client or server or both, it is necessary to add one or both of the following to the same file:

```
#define THERMOSTAT_SERVER
#define THERMOSTAT_CLIENT
```

Optional Attributes

The optional attributes for the Thermostat cluster (see Section 33.2) are enabled by defining:

- CLD_THERMOSTAT_ATTR_ID_LOCAL_TEMPERATURE
- CLD_THERMOSTAT_ATTR_ID_OUTDOOR_TEMPERATURE
- CLD_THERMOSTAT_ATTR_ID_OCCUPANCY
- CLD_THERMOSTAT_ATTR_ID_ABS_MIN_HEAT_SETPOINT_LIMIT
- CLD_THERMOSTAT_ATTR_ID_ABS_MAX_HEAT_SETPOINT_LIMIT
- CLD_THERMOSTAT_ATTR_ID_ABS_MIN_COOL_SETPOINT_LIMIT
- CLD_THERMOSTAT_ATTR_ID_ABS_MAX_COOL_SETPOINT_LIMIT
- CLD_THERMOSTAT_ATTR_ID_PI_COOLING_DEMAND
- CLD_THERMOSTAT_ATTR_ID_PI_HEATING_DEMAND
- CLD_THERMOSTAT_ATTR_ID_LOCAL_TEMPERATURE_CALIBRATION
- CLD_THERMOSTAT_ATTR_ID_OCCUPIED_COOLING_SETPOINT

- CLD_THERMOSTAT_ATTR_ID_OCCUPIED_HEATING_SETPOINT
- CLD_THERMOSTAT_ATTR_ID_UNOCCUPIED_COOLING_SETPOINT
- CLD_THERMOSTAT_ATTR_ID_UNOCCUPIED_HEATING_SETPOINT
- CLD_THERMOSTAT_ATTR_ID_MIN_HEAT_SETPOINT_LIMIT
- CLD_THERMOSTAT_ATTR_ID_MAX_HEAT_SETPOINT_LIMIT
- CLD_THERMOSTAT_ATTR_ID_MIN_COOL_SETPOINT_LIMIT
- CLD_THERMOSTAT_ATTR_ID_MAX_COOL_SETPOINT_LIMIT
- CLD_THERMOSTAT_ATTR_ID_MIN_SETPOINT_DEAD_BAND
- CLD_THERMOSTAT_ATTR_ID_REMOTE_SENSING
- CLD_THERMOSTAT_ATTR_ID_CONTROL_SEQUENCE_OF_OPERATION
- CLD_THERMOSTAT_ATTR_ID_SYSTEM_MODE
- CLD_THERMOSTAT_ATTR_ID_ALARM_MASK

Global Attributes

Add this line to enable the optional Attribute Reporting Status attribute:

```
#define CLD_THERMOSTAT_ATTR_ID_ATTRIBUTE_REPORTING_STATUS
```

Add this line to define the value (n) of the Cluster Revision attribute:

```
#define CLD_THERMOSTAT_CLUSTER_REVISION <n>
```

The default value is 1, which corresponds to the revision of the cluster in the ZCL r6 specification (see Section 2.4).

Minimum Cooling Setpoint

The value of the attribute `i16MinCoolSetpointLimit` can be set as follows:

```
#define CLD_THERMOSTAT_MIN_COOLING_SETPOINT          n
```

where n is the value to be set (in two’s complement form). The default value is 0x954D.

Maximum Cooling Setpoint

The value of the attribute `i16MaxCoolSetpointLimit` can be set as follows:

```
#define CLD_THERMOSTAT_MAX_COOLING_SETPOINT n
```

where n is the value to be set (in two’s complement form). The default value is 0x7FFF.

Minimum Heating Setpoint

The value of the attribute `i16MinHeatSetpointLimit` can be set as follows:

```
#define CLD_THERMOSTAT_MIN_HEATING_SETPOINT n
```

where n is the value to be set (in two’s complement form). The default value is 0x954D.

Maximum Heating Setpoint

The value of the attribute `i16MaxHeatSetpointLimit` can be set as follows:

```
#define CLD_THERMOSTAT_MAX_HEATING_SETPOINT n
```

where n is the value to be set (in two's complement form). The default value is 0x7FFF.

34 Fan Control Cluster

This chapter describes the Fan Control cluster which is defined in the ZCL.

The Fan Control cluster has a Cluster ID of 0x0202.

34.1 Overview

The Fan Control cluster is used to control the speed of a fan which may be part of a heating or cooling system. It allows the speed or state of the fan to be set, as well as the possible speeds/states that a thermostat can set.

To use the functionality of this cluster, you must include the file **FanControl.h** in your application and enable the cluster by defining CLD_FAN_CONTROL in the **zcl_options.h** file.

A Fan Control cluster instance can act as a client or a server. The inclusion of the client or server software must be pre-defined in the application's compile-time options (in addition, if the cluster is to reside on a custom endpoint then the role of client or server must also be specified when creating the cluster instance).

The compile-time options for the Fan Control cluster are fully detailed in Section 34.6.

34.2 Fan Control Structure and Attributes

The structure definition for the Fan Control cluster is shown below.

```
typedef struct
{
    #ifdef FAN_CONTROL_SERVER
        zenum8  e8FanMode;
        zenum8  e8FanModeSequence;
    #endif
    zuint16 u16ClusterRevision;
} tsCLD_FanControl;
```

where:

- `e8FanMode` is a server attribute that represents the current speed/state of the fan. The attribute can be set to one of the enumerated values listed in Section 34.5.2, representing off, low, medium, high, on, auto or smart.
- `e8FanModeSequence` is a server attribute that specifies the possible fan speeds/states that a thermostat can set. The attribute can be set to one of the enumerated values listed in Section 34.5.3, each representing a set of possible fan speeds/states.
- `u16ClusterRevision` is a mandatory attribute that specifies the revision of the cluster specification on which this cluster instance is based. The cluster specification in the ZCL r6 corresponds to a cluster revision of 1. The value is incremented by one for each subsequent revision of the cluster specification. For cluster specifications that pre-date the ZCL r6, this attribute is set to 0.

34.3 Initialisation

The function **eCLD_CreateFanControl()** is used to create an instance of the Fan Control cluster. The function is generally called by the initialisation function for the host device.

34.4 Functions

The following Fan Control cluster function is provided in the NXP implementation of the ZCL:

34.4.1 eCLD_CreateFanControl

```
teZCL_Status eCLD_CreateFanControl(
         tsZCL_ClusterInstance *psClusterInstance,
         bool_t bIsServer,
         tsZCL_ClusterDefinition *psClusterDefinition,
         void *pvEndPointSharedStructPtr,
        uint8 *pu8AttributeControlBits);
```

Description

This function creates an instance of the Fan Control 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 Fan Control cluster instance on the endpoint, but instances of other clusters may also be created on the same endpoint by calling their corresponding creation functions.

Note: *This function must not be called for an endpoint on which a standard ZigBee device is used. In this case, the device and its supported clusters must be registered on the endpoint using the relevant device registration function.*

When used, this function must be called after the stack has been started and after the application profile has been initialized.

Parameters

psClusterInstance Pointer to structure containing information about the cluster instance to be created (see Section 6.1.16). This structure is updated by the function by initialising individual structure fields.

bIsServer Type of cluster instance (server or client) to be created:

TRUE - server

FALSE - client

psClusterDefinition Pointer to structure indicating the type of cluster to be created (see Section 6.1.2). In this case, this structure must contain the details of the Fan Control cluster. This parameter can refer to a pre-filled structure called `tsCLD_FanControl` which is provided in the **FanControl.h** file.

pvEndPointSharedStructPtr Pointer to the shared structure used for attribute storage. This parameter should be the address of the structure of type `tsCLD_FanControl` which defines the attributes of the Fan Control cluster. The function initializes the attributes with default values.

pu8AttributeControlBits Pointer to an array of **uint8** values, with one element for each attribute in the cluster (see above).

Returns

E_ZCL_SUCCESS

E_ZCL_ERR_PARAMETER_NULL

34.5 Enumerations

34.5.1 teCLD_FanControl_AttributeID

The following structure contains the enumerations used to identify the server attributes of the Fan Control cluster.

```
typedef enum
{
    E_CLD_FAN_CONTROL_ATTR_ID_FAN_MODE              = 0x0000,
    E_CLD_FAN_CONTROL_ATTR_ID_FAN_MODE_SEQUENCE,
} teCLD_FanControl_AttributeID;
```

34.5.2 teCLD_FC_FanMode

The following structure contains the enumerations used to set the value of the `e8FanMode` attribute in the `tsCLD_FanControl` structure (see Section 34.2).

```
typedef enum
{
    E_CLD_FC_FAN_MODE_OFF       = 0x00,
    E_CLD_FC_FAN_MODE_LOW,      //0x01
    E_CLD_FC_FAN_MODE_MEDIUM,   //0x02
    E_CLD_FC_FAN_MODE_HIGH,     //0x03
    E_CLD_FC_FAN_MODE_ON,       //0x04
    E_CLD_FC_FAN_MODE_AUTO,     //0x05
    E_CLD_FC_FAN_MODE_SMART,    //0x06
} teCLD_FC_FanMode;
```

The above enumerations are described in the table below.

Table 53. 'Fan Mode' Enumerations

Enumeration	Description (Fan State/Speed)
E_CLD_FC_FAN_MODE_OFF	Off
E_CLD_FC_FAN_MODE_LOW	Low
E_CLD_FC_FAN_MODE_MEDIUM	Medium
E_CLD_FC_FAN_MODE_HIGH	High
E_CLD_FC_FAN_MODE_ON	On
E_CLD_FC_FAN_MODE_AUTO	Auto (fan speed is self-regulated)
E_CLD_FC_FAN_MODE_SMART	Smart (when the space is occupied, the fan is always on)

34.5.3 teCLD_FC_FanModeSequence

The following structure contains the enumerations used to set the value of the `e8FanModeSequence` attribute in the `tsCLD_FanControl` structure (see Section 34.2).

```
typedef enum
{
    E_CLD_FC_FAN_MODE_SEQUENCE_LOW_MED_HIGH         = 0x00,
    E_CLD_FC_FAN_MODE_SEQUENCE_LOW_HIGH,            //0x01
    E_CLD_FC_FAN_MODE_SEQUENCE_LOW_MED_HIGH_AUTO,   //0x02
    E_CLD_FC_FAN_MODE_SEQUENCE_LOW_HIGH_AUTO,       //0x03
```

```
    E_CLD_FC_FAN_MODE_SEQUENCE_ON_AUTO,              //0x04
} teCLD_FC_FanModeSequence;
```

The above enumerations are described in the table below (the fan speeds/states refer to those listed in Section 34.5.2).

Table 54. 'Fan Mode Sequence' Enumerations

Enumeration	Description (Set of Fan Speeds/States)
E_CLD_FC_FAN_MODE_SEQUENCE_LOW_MED_HIGH	Low/Med/High
E_CLD_FC_FAN_MODE_SEQUENCE_LOW_HIGH	Low/High
E_CLD_FC_FAN_MODE_SEQUENCE_LOW_MED_HIGH_AUTO	Low/Med/High/Auto
E_CLD_FC_FAN_MODE_SEQUENCE_LOW_HIGH_AUTO	Low/High/Auto
E_CLD_FC_FAN_MODE_SEQUENCE_ON_AUTO	On/Auto

34.6 Compile-time options

To enable the Fan Control cluster in the code to be built, it is necessary to add the following to the **zcl_options.h** file:

```
#define CLD_FAN_CONTROL
```

In addition, to include the software for a cluster client or server or both, it is necessary to add one or both of the following to the same file:

```
#define FAN_CONTROL_CLIENT
#define FAN_CONTROL_SERVER
```

35 Thermostat UI Configuration Cluster

This chapter outlines the Thermostat User Interface (UI) Configuration cluster which is defined in the ZCL and provides an interface for configuring the user interface (keypad and/or LCD screen) of a thermostat - this interface may be located on a controlling device which is remote from the thermostat.

The Thermostat UI Configuration cluster has a Cluster ID of 0x0204.

35.1 Overview

The Thermostat UI Configuration cluster is required in ZigBee devices as indicated in the table below.

Table 55. Thermostat UI Configuration Cluster in ZigBee Devices

	Server-side	Client-side
Mandatory in...		
Optional in...	Thermostat	Configuration Tool Combined Interface

The Thermostat UI Configuration cluster is enabled by defining CLD_THERMOSTAT_UI_CONFIG in the **zcl_options.h** file.

The inclusion of the client or server software must be pre-defined in the application's compile-time options (in addition, if the cluster is to reside on a custom endpoint then the role of client or server must also be specified when creating the cluster instance).

The compile-time options for the Thermostat UI Configuration cluster are fully detailed in Section 35.7.

35.2 Cluster structure and attributes

The Thermostat UI Configuration cluster is contained in the following `tsCLD_ThermostatUIConfig` structure:

```
typedef struct
{
#ifdef THERMOSTAT_UI_CONFIG_SERVER
    zenum8          eTemperatureDisplayMode;
    zenum8          eKeypadLockout;
#endif
    zuint16         u16ClusterRevision;
} tsCLD_ThermostatUIConfig;;
```

where:

- `eTemperatureDisplayMode` specifies the units (Celsius or Fahrenheit) used to display temperature on the screen of the user interface. Enumerations are provided:
  - E_CLD_THERMOSTAT_UI_CONFIG_TEMPERATURE_DISPLAY_MODE_CELSIUS
  - E_CLD_THERMOSTAT_UI_CONFIG_TEMPERATURE_DISPLAY_MODE_FAHRENHEIT
- `eKeypadLockout` specifies the level of functionality that is available via the keypad of the user interface. Enumerations are provided:
  - E_CLD_THERMOSTAT_UI_CONFIG_KEYPAD_LOCKOUT_NO_LOCKOUT
  - E_CLD_THERMOSTAT_UI_CONFIG_KEYPAD_LOCKOUT_LEVEL_1_LOCKOUT
  - E_CLD_THERMOSTAT_UI_CONFIG_KEYPAD_LOCKOUT_LEVEL_2_LOCKOUT
  - E_CLD_THERMOSTAT_UI_CONFIG_KEYPAD_LOCKOUT_LEVEL_3_LOCKOUT
  - E_CLD_THERMOSTAT_UI_CONFIG_KEYPAD_LOCKOUT_LEVEL_4_LOCKOUT

  - E_CLD_THERMOSTAT_UI_CONFIG_KEYPAD_LOCKOUT_LEVEL_5_LOCKOUT
    The functionality of each level is manufacturer-defined but level 5 represents the minimum functionality.
- `u16ClusterRevision` is a mandatory attribute that specifies the revision of the cluster specification on which this cluster instance is based. The cluster specification in the ZCL r6 corresponds to a cluster revision of 1. The value is incremented by one for each subsequent revision of the cluster specification. This attribute is also described in Section 2.4.

35.3 Initialization

The function **eCLD_ThermostatUIConfigCreateThermostatUIConfig()** is used to create an instance of the Thermostat UI Configuration cluster. The function is generally called by the initialization function for the host device.

35.4 Functions

The following Thermostat UI Configuration cluster functions are provided:

35.4.1 eCLD_ThermostatUIConfigCreateThermostatUIConfig

```
teZCL_Status eCLD_ThermostatUIConfigCreateThermostatUIConfig(
  tsZCL_ClusterInstance *psClusterInstance,
  bool_t bIsServer,
  sZCL_ClusterDefinition *psClusterDefinition,
  void *pvEndPointSharedStructPtr,
  uint8 *pu8AttributeControlBits);
```

Description

This function creates an instance of the Thermostat UI Configuration 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 Thermostat UI Configuration cluster instance on the endpoint, but instances of other clusters may also be created on the same endpoint by calling their corresponding creation functions.

Note: *This function must not be called for an endpoint on which a standard ZigBee device (for example, the Thermostat device) is used. In this case, the device and its supported clusters must be registered on the endpoint using the relevant device registration function.*

When used, this function must be the first Thermostat UI Configuration cluster function called in the application, and must be called after the stack has been started and after the ZCL has been initialized.

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 Thermostat UI Configuration cluster.

The function initializes the array elements to zero.

Parameters

- *psClusterInstance*: Pointer to structure containing information about the cluster instance to be created (see Section 6.1.16). This structure is updated by the function by initializing individual structure fields.
- *bIsServer* : Type of cluster instance (server or client) to be created:
- TRUE - server
- FALSE - client
- *psClusterDefinition*: Pointer to structure indicating the type of cluster to be created (see Section 6.1.2). In this case, this structure must contain the details of the Thermostat UI Configuration cluster. This parameter can refer to a pre-filled structure called `sCLD_ThermostatUIConfig` which is provided in the **ThermostatUIConfig.h** file.
- *pvEndPointSharedStructPtr*: Pointer to the shared structure used for attribute storage. This parameter should be the address of the structure of type `tsCLD_ThermostatUIConfig` which defines the attributes of Thermostat UI Configuration cluster. The function initializes the attributes with default values.
- *pu8AttributeControlBits*: Pointer to an array of **uint8** values, with one element for each attribute in the cluster (see above).

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_INVALID_VALUE

35.4.2 eCLD_ThermostatUIConfigConvertTemp

```
teZCL_Status eCLD_ThermostatUIConfigConvertTemp(
     uint8 u8SourceEndPointId,
     bool bConvertCToF,
     int16 *pi16Temperature);
```

Description

This function can be used on a Thermostat UI Configuration cluster server to convert a temperature from units of Celsius to Fahrenheit or vice-versa (the direction must be specified). The temperature value to be converted is provided to the function as a pointer to a memory location where the input value is stored. This stored value is replaced with the converted temperature value by the function (over-writing the input value).

Parameters

- *u8SourceEndPointId*: Number of the endpoint on which the Thermostat UI Configuration cluster resides
- *bConvertCToF*: Direction of temperature conversion:
- TRUE - Celsius to Fahrenheit
- FALSE - Fahrenheit to Celsius
- *pi16Temperature*: Pointer to location containing the temperature value to be converted. The converted temperature value is also output to this location by the function

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_INVALID_VALUE

- E_ZCL_DENY_ATTRIBUTE_ACCESS
- E_ZCL_ERR_CLUSTER_NOT_FOUND

35.5 Return codes

The Thermostat UI Configuration cluster functions use the ZCL return codes defined in Section 7.2.

35.6 Enumerations

35.6.1 'Attribute ID' Enumerations

The following structure contains the enumerations used to identify the attributes of the Thermostat UI Configuration cluster.

```
typedef enum
{
  E_CLD_THERMOSTAT_UI_CONFIG_ATTR_ID_TEMPERATURE_DISPLAY_MODE = 0x0000
  E_CLD_THERMOSTAT_UI_CONFIG_ATTR_ID_KEYPAD_LOCKOUT
} teCLD_ThermostatUIConfig_AttributeID;
```

35.6.2 'Temperature Display Mode' Enumerations

The following enumerations are used to set the optional attribute `eTemperatureDisplayMode` in the Thermostat UI Configuration cluster structure `tsCLD_ThermostatUIConfig`.

```
typedef enum
{
  E_CLD_THERMOSTAT_UI_CONFIG_TEMPERATURE_DISPLAY_MODE_CELSIUS = 0x00,
  E_CLD_THERMOSTAT_UI_CONFIG_TEMPERATURE_DISPLAY_MODE_FAHRENHEIT
} teCLD_ThermostatUIConfig_TemperatureDisplay;
```

The above enumerations represent the units of temperature available to display temperature on the screen of the user interface and are described in the table below.

Table 56. 'Temperature Display Mode' Enumerations

Enumeration	Description
E_CLD_THERMOSTAT_UI_CONFIG_TEMPERATURE_DISPLAY_MODE_CELSIUS	Display temperature in Celsius
E_CLD_THERMOSTAT_UI_CONFIG_TEMPERATURE_DISPLAY_MODE_FAHRENHEIT	Display temperature in Fahrenheit

35.6.3 'Keypad Functionality' Enumerations

The following enumeration is used to set the optional attribute `eKeypadLockout` in the Thermostat UI Configuration cluster structure `tsCLD_ThermostatUIConfig`.

```
typedef enum
{
    E_CLD_THERMOSTAT_UI_CONFIG_KEYPAD_LOCKOUT_NO_LOCKOUT = 0x00,
    E_CLD_THERMOSTAT_UI_CONFIG_KEYPAD_LOCKOUT_LEVEL_1_LOCKOUT,
    E_CLD_THERMOSTAT_UI_CONFIG_KEYPAD_LOCKOUT_LEVEL_2_LOCKOUT,
    E_CLD_THERMOSTAT_UI_CONFIG_KEYPAD_LOCKOUT_LEVEL_3_LOCKOUT,
    E_CLD_THERMOSTAT_UI_CONFIG_KEYPAD_LOCKOUT_LEVEL_4_LOCKOUT,
    E_CLD_THERMOSTAT_UI_CONFIG_KEYPAD_LOCKOUT_LEVEL_5_LOCKOUT
} teCLD_ThermostatUIConfig_KeyPadLockout;
```

The above enumerations represent levels of functionality available via the keypad of the user interface. The functionality of each level is manufacturer-defined but level 5 represents the minimum functionality.

35.7 Compile-time Options

To enable the Thermostat UI Configuration cluster in the code to be built, it is necessary to add the following to the **zcl_options.h** file:

```
#define CLD_THERMOSTAT_UI_CONFIG
```

In addition, to include the software for a cluster client or server or both, it is necessary to add one or both of the following to the same file:

```
#define THERMOSTAT_UI_CONFIG_SERVER
#define THERMOSTAT_UI_CONFIG_CLIENT
```

Add this line to define the value (n) of the Cluster Revision attribute:

```
#define CLD_THERMOSTAT_UI_CONFIG_CLUSTER_REVISION <n>
```

The default value is 1, which corresponds to the revision of the cluster in the ZCL r6 specification (see Section 2.4).

Part VII: Closure Clusters

This part comprises one chapter:

- Chapter 36 details the **Door Lock** cluster

36 Door Lock Cluster

This chapter outlines the Door Lock cluster, which provides an interface to a set values representing the state of a door lock and (optionally) the door.

The Door Lock cluster has a Cluster ID of 0x0101.

36.1 Overview

The Door Lock cluster is required in ZigBee devices as indicated in the table below.

Table 57. Door Lock Cluster in ZigBee Devices

	Server-side	Client-side
Mandatory in...	Door Lock	Door Lock Controller
Optional in...		Remote Control

The Door Lock cluster is enabled by defining CLD_DOOR_LOCK in the **zcl_options.h** file.

The inclusion of the client or server software must be pre-defined in the application's compile-time options (in addition, if the cluster is to reside on a custom endpoint then the role of client or server must also be specified when creating the cluster instance).

The compile-time options for the Door Lock cluster are fully detailed in Section 36.9.

36.2 Door Lock Cluster structure and attributes

The Door Lock cluster is contained in the following `tsCLD_DoorLock` structure:

```
typedef struct
{
#ifdef DOOR_LOCK_SERVER
    zenum8                  eLockState;
    zenum8                  eLockType;
    zbool                   bActuatorEnabled;
#ifdef CLD_DOOR_LOCK_ATTR_DOOR_STATE
    zenum8                  eDoorState;
#endif
#ifdef CLD_DOOR_LOCK_ATTR_NUMBER_OF_DOOR_OPEN_EVENTS
    zuint32                  u32NumberOfDoorOpenEvent;
#endif
#ifdef CLD_DOOR_LOCK_ATTR_NUMBER_OF_DOOR_CLOSED_EVENTS
    zuint32                  u32NumberOfDoorClosedEvent;
#endif
#ifdef CLD_DOOR_LOCK_ATTR_NUMBER_OF_MINUTES_DOOR_OPENED
    zuint16                  u16NumberOfMinutesDoorOpened;
#endif
#ifdef CLD_DOOR_LOCK_ZIGBEE_SECUTRITY_LEVEL
    zuint8                    u8ZigbeeSecurityLevel;
#endif
#ifdef CLD_DOOR_LOCK_ATTRIBUTE_REPORTING_STATUS
    zuint8                    u8AttributeReportingStatus;
#endif
#endif
    zuint16                   u16ClusterRevision;
} tsCLD_DoorLock;
```

where:

- `eLockState` is a mandatory attribute indicating the state of the lock, one of:
  - E_CLD_DOORLOCK_LOCK_STATE_NOT_FULLY_LOCKED
  - E_CLD_DOORLOCK_LOCK_STATE_LOCK
  - E_CLD_DOORLOCK_LOCK_STATE_UNLOCK
- `eLockType` is a mandatory attribute representing the type of door lock, one of:
  - E_CLD_DOORLOCK_LOCK_TYPE_DEAD_BOLT
  - E_CLD_DOORLOCK_LOCK_TYPE_MAGNETIC
  - E_CLD_DOORLOCK_LOCK_TYPE_OTHER
  - E_CLD_DOORLOCK_LOCK_TYPE_MORTISE
  - E_CLD_DOORLOCK_LOCK_TYPE_RIM
  - E_CLD_DOORLOCK_LOCK_TYPE_LATCH_BOLT
  - E_CLD_DOORLOCK_LOCK_TYPE_CYLINDRICAL_LOCK
  - E_CLD_DOORLOCK_LOCK_TYPE_TUBULAR_LOCK
  - E_CLD_DOORLOCK_LOCK_TYPE_INTERCONNECTED_LOCK
  - E_CLD_DOORLOCK_LOCK_TYPE_DEAD_LATCH
  - E_CLD_DOORLOCK_LOCK_TYPE_DOOR_FURNITURE
- `bActuatorEnabled` is a mandatory attribute indicating whether the actuator for the door lock is enabled:
  - TRUE - enabled
  - FALSE - disabled
- `eDoorState` is an optional attribute indicating the current state of the door, one of:
  - E_CLD_DOORLOCK_DOOR_STATE_OPEN
  - E_CLD_DOORLOCK_DOOR_STATE_CLOSED
  - E_CLD_DOORLOCK_DOOR_STATE_ERROR_JAMMED
  - E_CLD_DOORLOCK_DOOR_STATE_ERROR_FORCED_OPEN
  - E_CLD_DOORLOCK_DOOR_STATE_ERROR_UNSPECIFIED
- `u32NumberOfDoorOpenEvent` is an optional attribute representing the number of ‘door open’ events that have occurred
- `u32NumberOfDoorClosedEvent` is an optional attribute representing the number of ‘door close’ events that have occurred
- `u16NumberOfMinutesDoorOpened` is an optional attribute representing the length of time, in minutes, that the door has been open since the last ‘door open’ event
- `u8ZigbeeSecurityLevel` is an optional attribute representing the ZigBee PRO security level that should be applied to communications between a cluster server and client:
  - 0: Network-level security only
  - 1 or higher: Application-level security (in addition to Network-level security)

    Application-level security is an enhancement to the Door Lock cluster and is currently not certifiable.

Note: *The application must not write directly to the* `u8ZigbeeSecurityLevel` *attribute. If required, Application-level security should be enabled only using the function* ***eCLD_DoorLockSetSecurityLevel()****. For more information, refer to the description of this function on page 771.*

```
u8AttributeReportingStatus is an optional attribute that should be enabled when attribute
u16ClusterRevision is a mandatory attribute that specifies the revision of the cluster spe
```

36.3 Attributes for Default Reporting

The following attributes of the Door Lock cluster can be selected for default reporting:

```
eLockState
eDoorState
```

Attribute reporting (including default reporting) is described in Appendix B. Enabling reports for these attributes is described in Appendix B.3.6.

36.4 Door Lock Events

The Door Lock cluster has its own events that are handled through the callback mechanism outlined in Chapter 3. If a device uses the Door Lock cluster then Door Lock 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 **eHA_RegisterDoorLockEndPoint()** for a Door Lock device). The relevant callback function will then be invoked when a Door Lock event occurs.

For a Door Lock 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 `tsCLD_DoorLockCallBackMessage` structure:

```
typedef struct
{
    uint8    u8CommandId;
    union
    {
       tsCLD_DoorLock_LockUnlockResponsePayload *psLockUnlockResponsePayload;
    }uMessage;
 }tsCLD_DoorLockCallBackMessage;
```

When a Door Lock event occurs, one of two command types could have been received. The relevant command type is specified through the `u8CommandId` field of the `tsCLD_DoorLockCallBackMessage` structure. The possible command types are detailed below.

Table 58. Door Lock Command Types

u8CommandId Enumeration	Description
E_CLD_DOOR_LOCK_CMD_LOCK	A lock request command has been received by the clus-ter server
E_CLD_DOOR_LOCK_CMD_UNLOCK	An unlock request command has been received by the cluster server

36.5 Functions

The following Door Lock cluster functions are provided:

36.5.1 eCLD_DoorLockCreateDoorLock

```
teZCL_Status eCLD_DoorLockCreateDoorLock(
        tsZCL_ClusterInstance *psClusterInstance,
        bool_t bIsServer,
        tsZCL_ClusterDefinition                         *psClusterDefinition,
        void *pvEndPointSharedStructPtr,
        uint8 *pu8AttributeControlBits);
```

Description

This function creates an instance of the Door Lock 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 Door Lock cluster instance on the endpoint, but instances of other clusters may also be created on the same endpoint by calling their corresponding creation functions.

Note: *This function must not be called for an endpoint on which a standard ZigBee device (e.g. the Door Lock device) will be used. In this case, the device and its supported clusters must be registered on the endpoint using the relevant device registration function.*

When used, this function must be the first Door Lock cluster function called in the application, and must be called after the stack has been started and after the ZCL has been initialized.

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 Door Lock cluster.

The function initializes the array elements to zero.

Parameters

psClusterInstance Pointer to structure containing information about the cluster instance to be created (see Section 6.1.16). This structure is updated by the function by initialising individual structure fields.

bIsServer Type of cluster instance (server or client) to be created:

TRUE - server

FALSE - client

psClusterDefinition Pointer to structure indicating the type of cluster to be created (see Section 6.1.2). In this case, this structure must contain the details of the Door Lock cluster. This parameter can refer to a pre-filled structure called `sCLD_DoorLock` which is provided in the **DoorLock.h** file.

pvEndPointSharedStructPtr Pointer to the shared structure used for attribute storage. This parameter should be the address of the structure of type `tsCLD_DoorLock` which defines the attributes of Door Lock cluster. The function initializes the attributes with default values.

pu8AttributeControlBits Pointer to an array of **uint8** values, with one element for each attribute in the cluster (see above).

Returns

E_ZCL_SUCCESS
E_ZCL_FAIL
E_ZCL_ERR_PARAMETER_NULL

E_ZCL_ERR_INVALID_VALUE

36.5.2 eCLD_DoorLockSetLockState

```
teZCL_Status eCLD_DoorLockSetLockState(
        uint8 u8SourceEndPointId,
        teCLD_DoorLock_LockState eLock);
```

Description

This function can be used on a Door Lock cluster server to set the value of the `eLockState` attribute which represents the current state of the door lock (locked, unlocked or not fully locked).

Depending on the specified value of *eLock*, the attribute will be set to one of the following:

- E_CLD_DOORLOCK_LOCK_STATE_NOT_FULLY_LOCKED
- E_CLD_DOORLOCK_LOCK_STATE_LOCK
- E_CLD_DOORLOCK_LOCK_STATE_UNLOCK

This function generates an update event to inform the application when the change has been made.

Parameters

u8SourceEndPointId Number of the endpoint on which the Door Lock cluster resides
eLock State in which to put the door lock, one of:
E_CLD_DOORLOCK_LOCK_STATE_NOT_FULLY_LOCKED
E_CLD_DOORLOCK_LOCK_STATE_LOCK
E_CLD_DOORLOCK_LOCK_STATE_UNLOCK

Returns

E_ZCL_SUCCESS
E_ZCL_FAIL

36.5.3 eCLD_DoorLockGetLockState

```
teZCL_Status eCLD_DoorLockGetLockState(
        uint8 u8SourceEndPointId,
        teCLD_DoorLock_LockState *peLock);
```

Description

This function can be used on a Door Lock cluster server to obtain the value of the `eLockState` attribute which represents the current state of the door lock (locked, unlocked or not fully locked).

The value of the attribute is returned through the location pointed to by `peLock` and can be any one of the following:

- E_CLD_DOORLOCK_LOCK_STATE_NOT_FULLY_LOCKED
- E_CLD_DOORLOCK_LOCK_STATE_LOCK
- E_CLD_DOORLOCK_LOCK_STATE_UNLOCK

Parameters

u8SourceEndPointId Number of the endpoint on which the Door Lock cluster resides
peLock Pointer to location to receive the obtained state of the door lock, which will be one of:
E_CLD_DOORLOCK_LOCK_STATE_NOT_FULLY_LOCKED
E_CLD_DOORLOCK_LOCK_STATE_LOCK
E_CLD_DOORLOCK_LOCK_STATE_UNLOCK

Returns

E_ZCL_SUCCESS
E_ZCL_FAIL

36.5.4 eCLD_DoorLockCommandLockUnlockRequestSend

```
teZCL_Status eCLD_DoorLockCommandLockUnlockRequestSend(
        uint8 u8SourceEndPointId,
        uint8 u8DestinationEndPointId,
        tsZCL_Address *psDestinationAddress,
        uint8 *pu8TransactionSequenceNumber,
        teCLD_DoorLock_CommandID eCommand);
```

Description

This function can be used on a Door Lock cluster client to send a lock or unlock command to the Door Lock cluster server.

A pointer must be specified to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is 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 is sent
u8DestinationEndPointId Number of the remote endpoint to which the request is sent
psDestinationAddress Pointer to a structure containing the address of the remote node to which the request is sent
pu8TransactionSequenceNumber Pointer to a location to store the Transaction Sequence Number (TSN) of the request
eCommand The command to be sent, one of:
E_CLD_DOOR_LOCK_CMD_LOCK
E_CLD_DOOR_LOCK_CMD_UNLOCK

Returns

E_ZCL_SUCCESS
E_ZCL_ERR_PARAMETER_NULL
E_ZCL_ERR_EP_RANGE
E_ZCL_ERR_EP_UNKNOWN
E_ZCL_ERR_CLUSTER_NOT_FOUND
E_ZCL_ERR_ZBUFFER_FAIL

E_ZCL_ERR_ZTRANSMIT_FAIL

36.5.5 eCLD_DoorLockSetSecurityLevel

```
teZCL_Status eCLD_DoorLockSetSecurityLevel(
                uint8 u8SourceEndPointId,
                bool bServer,
                uint8 u8SecurityLevel);
```

Description

This function can be used to set the level of security to be used by the Door Lock cluster: Network-level security or Application-level security. By default, only Network-level security is implemented, but this function can be used to enable Application-level security (in addition to Network-level security). For more information on ZigBee security, refer to the *ZigBee 3.0 Stack User Guide (JNUG3130)*.

Application-level security is an enhancement to the Door Lock cluster and is currently not certifiable. It is enabled through an optional attribute of the cluster, but the application must not write directly to this attribute - if required, Application-level security should be enabled only using this function.

To use Application-level security, it is necessary to call this function on the Door Lock cluster server and client nodes. If an application link key is to be used which is not the default one, the new link key must be subsequently specified on both nodes using the ZigBee PRO function **ZPS_eAplZdoAddReplaceLinkKey()**.

Parameters

u8SourceEndPointId Number of the local endpoint on which the Door Lock cluster resides
bIsServer Type of local cluster instance (server or client):
TRUE - server
FALSE - client
u8SecurityLevel The security level to be set:
0: Network-level security only
1 or higher: Application-level security

Returns

E_ZCL_SUCCESS
E_ZCL_FAIL

36.6 Return codes

The Door Lock cluster functions use the ZCL return codes defined in Section 7.2.

36.7 Enumerations

36.7.1 'Attribute ID' Enumerations

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

```
typedef enum
{
    E_CLD_DOOR_LOCK_ATTR_ID_LOCK_STATE = 0x0000,
    E_CLD_DOOR_LOCK_ATTR_ID_LOCK_TYPE,
```

```
    E_CLD_DOOR_LOCK_ATTR_ID_ACTUATOR_ENABLED,
    E_CLD_DOOR_LOCK_ATTR_ID_DOOR_STATE,
    E_CLD_DOOR_LOCK_ATTR_ID_NUMBER_OF_DOOR_OPEN_EVENTS,
    E_CLD_DOOR_LOCK_ATTR_ID_NUMBER_OF_DOOR_CLOSED_EVENTS,
    E_CLD_DOOR_LOCK_ATTR_ID_NUMBER_OF_MINUTES_DOOR_OPENED,
    E_CLD_DOOR_LOCK_ATTR_ID_ZIGBEE_SECURITY_LEVEL = 0x0034
} teCLD_DoorLock_Cluster_AttrID;
```

36.7.2 'Lock State' Enumerations

The following enumerations are used to set the `eLockState` element in the Door Lock cluster structure `tsCLD_DoorLock`.

```
typedef enum
{
    E_CLD_DOORLOCK_LOCK_STATE_NOT_FULLY_LOCKED  = 0x00,
    E_CLD_DOORLOCK_LOCK_STATE_LOCKED,
    E_CLD_DOORLOCK_LOCK_STATE_UNLOCKED,
    E_CLD_DOORLOCK_LOCK_STATE_UNDEFINED         = 0xFF
} teCLD_DoorLock_LockState;
```

The above enumerations are described in the table below.

Table 59. 'Lock State' Enumerations

Enumeration	Description
E_CLD_DOORLOCK_LOCK_STATE_NOT_FULLY_LOCKED	Not fully locked
E_CLD_DOORLOCK_LOCK_STATE_LOCK	Locked
E_CLD_DOORLOCK_LOCK_STATE_UNLOCK	Unlocked

36.7.3 'Lock Type' Enumerations

The following enumerations are used to set the `eLockType` element in the Door Lock cluster structure `tsCLD_DoorLock`.

```
typedef enum
{
    E_CLD_DOORLOCK_LOCK_TYPE_DEAD_BOLT  = 0x00,
    E_CLD_DOORLOCK_LOCK_TYPE_MAGNETIC,
    E_CLD_DOORLOCK_LOCK_TYPE_OTHER,
    E_CLD_DOORLOCK_LOCK_TYPE_MORTISE,
    E_CLD_DOORLOCK_LOCK_TYPE_RIM,
    E_CLD_DOORLOCK_LOCK_TYPE_LATCH_BOLT,
    E_CLD_DOORLOCK_LOCK_TYPE_CYLINDRICAL_LOCK,
    E_CLD_DOORLOCK_LOCK_TYPE_TUBULAR_LOCK,
    E_CLD_DOORLOCK_LOCK_TYPE_INTERCONNECTED_LOCK,
    E_CLD_DOORLOCK_LOCK_TYPE_DEAD_LATCH,
    E_CLD_DOORLOCK_LOCK_TYPE_DOOR_FURNITURE
} teCLD_DoorLock_LockType;
```

The above enumerations are described in the table below.

Table 60. 'Lock Type' Enumerations

Enumeration	Description
E_CLD_DOORLOCK_LOCK_TYPE_DEAD_BOLT	Dead bold lock

Table 60. 'Lock Type' Enumerations*...continued*

Enumeration	Description
E_CLD_DOORLOCK_LOCK_TYPE_MAGNETIC	Magnetic lock
E_CLD_DOORLOCK_LOCK_TYPE_OTHER	Other type of lock
E_CLD_DOORLOCK_LOCK_TYPE_MORTISE	Mortise lock
E_CLD_DOORLOCK_LOCK_TYPE_RIM	Rim lock
E_CLD_DOORLOCK_LOCK_TYPE_LATCH_BOLT	Latch bolt
E_CLD_DOORLOCK_LOCK_TYPE_CYLINDRICAL_LOCK	Cylindrical lock
E_CLD_DOORLOCK_LOCK_TYPE_TUBULAR_LOCK	Tubular lock
E_CLD_DOORLOCK_LOCK_TYPE_INTERCONNECTED_LOCK	Interconnected lock
E_CLD_DOORLOCK_LOCK_TYPE_DEAD_LATCH	Dead latch
E_CLD_DOORLOCK_LOCK_TYPE_DOOR_FURNITURE	Door furniture lock

36.7.4 'Door State' Enumerations

The following enumerations are used to set the optional `eDoorState` element in the Door Lock cluster structure `tsCLD_DoorLock`.

```
typedef enum
{
    E_CLD_DOORLOCK_DOOR_STATE_OPEN  = 0x00,
    E_CLD_DOORLOCK_DOOR_STATE_CLOSED,
    E_CLD_DOORLOCK_DOOR_STATE_ERROR_JAMMED,
    E_CLD_DOORLOCK_DOOR_STATE_ERROR_FORCED_OPEN,
    E_CLD_DOORLOCK_DOOR_STATE_ERROR_UNSPECIFIED,
    E_CLD_DOORLOCK_DOOR_STATE_UNDEFINED = 0xFF
} teCLD_DoorLock_DoorState;;
```

The above enumerations are described in the table below.

Table 61. 'Door State' Enumerations

Enumeration	Description
E_CLD_DOORLOCK_DOOR_STATE_OPEN	Door is open
E_CLD_DOORLOCK_DOOR_STATE_CLOSED	Door is closed
E_CLD_DOORLOCK_DOOR_STATE_ERROR_JAMMED	Door is jammed
E_CLD_DOORLOCK_DOOR_STATE_ERROR_FORCED_OPEN	Door has been forced open
E_CLD_DOORLOCK_DOOR_STATE_ERROR_UNSPECIFIED	Door is in an unknown state

36.7.5 'Command ID' Enumerations

The following enumerations are used to set specify the type of command (lock or unlock) sent to a Door Lock cluster server.

```
typedef enum
{
    E_CLD_DOOR_LOCK_CMD_LOCK
    E_CLD_DOOR_LOCK_CMD_UNLOCK
} teCLD_DoorLock_CommandID;
```

The above enumerations are described in the table below.

Table 62. 'Command ID' Enumerations

Enumeration	Description
E_CLD_DOOR_LOCK_CMD_LOCK	A lock command
E_CLD_DOOR_LOCK_CMD_UNLOCK	An unlock command

36.8 Structures

36.8.1 tsCLD_DoorLockCallBackMessage

For a Door Lock 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 `tsCLD_DoorLockCallBackMessage` structure:

```
typedef struct
{
    uint8 u8CommandId;
    union
    {
      tsCLD_DoorLock_LockUnlockResponsePayload *psLockUnlockResponsePayload;
    }uMessage;
 }tsCLD_DoorLockCallBackMessage;
```

where:

- `u8CommandId` indicates the type of Door Lock command (lock or unlock) that has been received, one of:
  - E_CLD_DOOR_LOCK_CMD_LOCK
  - E_CLD_DOOR_LOCK_CMD_UNLOCK
- `uMessage` is a union containing the command payload in the following form:
  - `psLockUnlockResponsePayload` is a pointer to a structure containing the response payload of the received command - see Section 36.8.2

36.8.2 tsCLD_DoorLock_LockUnlockResponsePayload

This stucture contains the payload of a lock/unlock command response (from the cluster server).

```
typedef struct
{
    zenum8     eStatus;
}tsCLD_DoorLock_LockUnlockResponsePayload;
```

where `eStatus` indicates whether the command was received:

0x00 - SUCCESS, 0x01 - FAILURE (all other values are reserved).

36.9 Compile-time options

To enable the Door Lock cluster in the code to be built, it is necessary to add the following to the **zcl_options.h** file:

```
#define CLD_DOOR_LOCK
```

In addition, to include the software for a cluster client or server or both, it is necessary to add one or both of the following to the same file:

```
#define CLD_DOOR_LOCK_SERVER
#define CLD_DOOR_LOCK_CLIENT
```

Optional Attributes

Add this line to enable the optional Door State attribute:

```
#define CLD_DOOR_LOCK_ATTR_DOOR_STATE
```

Add this line to enable the optional Number Of Door Open Events attribute:

```
#define CLD_DOOR_LOCK_ATTR_NUMBER_OF_DOOR_OPEN_EVENTS
```

Add this line to enable the optional Number Of Door Closed Events attribute:

```
#define CLD_DOOR_LOCK_ATTR_NUMBER_OF_DOOR_CLOSED_EVENTS
```

Add this line to enable the optional Number Of Minutes Door Opened attribute:

```
#define CLD_DOOR_LOCK_ATTR_NUMBER_OF_MINUTES_DOOR_OPENED
```

Add this line to enable the optional ZigBee Security Level attribute:

```
#define CLD_DOOR_LOCK_ZIGBEE_SECURITY_LEVEL
```

Add this line to enable the optional Attribute Reporting Status attribute:

```
#define CLD_DOOR_LOCK_ATTR_ID_ATTRIBUTE_REPORTING_STATUS
```

Global Attributes

Add this line to define the value (n) of the Cluster Revision attribute:

```
#define CLD_DOOR_LOCK_CLUSTER_REVISION <n>
```

The default value is 1, which corresponds to the revision of the cluster in the ZCL r6 specification (see Section 2.4).

Part VIII: Security and Safety Clusters

This part comprises three chapters:

- Chapter 37 details the **IAS Zone** cluster
- Chapter 38 details the **IAS ACE (Ancillary Control Equipment)** cluster
- Chapter 39 details the **IAS WD (Warning Device)** cluster

37 IAS Zone Cluster

This chapter describes the IAS Zone cluster which provides an interface to an IAS Zone device in an IAS (Intruder Alarm System).

The IAS Zone cluster has a Cluster ID of 0x0500.

37.1 Overview

The IAS Zone cluster provides an interface to an IAS Zone device, which provides security alarm triggers for a zone or region of a building (e.g. fire detection). The cluster allows an IAS Zone device to be configured/ controlled from a CIE (Control and Indicating Equipment) device. The server side of the cluster is implemented on the IAS Zone device and the client side is implemented on the CIE device. The IAS Zone device is detailed in the *ZigBee Devices User Guide (JNUG3131)*.

The cluster supports the following functionality:

- Up to two alarm types per zone, Alarm1 and Alarm2
- 'Low battery' reports
- Supervision of the IAS network

To use the functionality of this cluster, you must include the file **IASZone.h** in your application and enable the cluster by defining CLD_IASZONE in the **zcl_options.h** file.

The inclusion of the client or server software must be pre-defined in the application's compile-time options (in addition, if the cluster is to reside on a custom endpoint then the role of client or server must also be specified when creating the cluster instance).

The compile-time options for the IAS Zone cluster are fully detailed in Section 37.7.

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

- Zone information
- Zone settings

37.2 IAS Zone Structure and Attributes

The structure definition for the IAS Zone cluster is:

```
typedef struct
{
#ifdef IASZONE_SERVER
    zenum8     e8ZoneState;
    zenum16    e16ZoneType;
    zbmap16    b16ZoneStatus;
    zuint64    u64IASCIEAddress;
    zuint8     u8ZoneId;
#ifdef CLD_IASZONE_ATTR_ID_NUMBER_OF_ZONE_SENSITIVITY_LEVELS
    zuint8     u8NumberOfZoneSensitivityLevels;
#endif
#ifdef CLD_IASZONE_ATTR_ID_CURRENT_ZONE_SENSITIVITY_LEVEL
    zuint8     u8CurrentZoneSensitivityLevel;
#endif
#endif
    zuint16    u16ClusterRevision;
} tsCLD_IASZone;
```

where:

‘Zone Information’ Attribute Set

- `e8ZoneState` is a mandatory attribute which indicates the membership status of the device in an IAS system (enrolled or not enrolled) - one of:
  - E_CLD_IASZONE_STATE_NOT_ENROLLED (0x00)
  - E_CLD_IASZONE_STATE_ENROLLED (0x01)

    ‘Enrolled’ means that the cluster client will react to Zone State Change Notification commands from the cluster server.
- `e16ZoneType` is a mandatory attribute which indicates the zone type and the types of security detectors that can trigger the alarms, Alarm1 and Alarm2:

Table 63. e16ZoneType attribute description

Enumeration	Value	Type	Alarm1	Alarm2
E_CLD_IASZONE_TYPE_STANDARD_CIE	0x0000	Standard CIE	System alarm	-
E_CLD_IASZONE_TYPE_MOTION_ SENSOR	0x000D	Motion sensor	Intrusion indication	Presence indication
E_CLD_IASZONE_TYPE_CONTACT_ SWITCH	0x0015	Contact switch	First portal open close	Second portal open-close
E_CLD_IASZONE_TYPE_FIRE_SENSOR	0x0028	Fire sensor	Fire indication	-
E_CLD_IASZONE_TYPE_WATER_SENSOR	0x002A	Water sensor	Water overflow indication	-
E_CLD_IASZONE_TYPE_GAS_SENSOR	0x002B	Gas sensor	Carbon monoxide indication	Cooking indication
E_CLD_IASZONE_TYPE_PERSONAL_ EMERGENCY_DEVICE	0x002C	Personal emergency device	Fall/concussion	Emergency button
E_CLD_IASZONE_TYPE_VIBRATION_ MOVEMENT_SENSOR	0x002D	Vibration movement sensor	Movement indication	Vibration
E_CLD_IASZONE_TYPE_REMOTE_ CONTROL	0x010F	Remote control	Panic	Emergency
E_CLD_IASZONE_TYPE_KEY_FOB	0x0115	Key fob	Panic	Emergency
E_CLD_IASZONE_TYPE_KEYPAD	0x021D	Keypad	Panic	Emergency
E_CLD_IASZONE_TYPE_STANDARD_ WARNING_DEVICE	0x0225	Standard warning device	-	-
E_CLD_IASZONE_TYPE_INVALID_ZONE	0xFFFF	Invalid zone type	-	-

- `b16ZoneStatus` is a mandatory attribute which is a 16-bit bitmap indicating the status of each of the possible notification triggers from the device:

Table 64. 16ZoneStatus options

Bit	Description
0	Alarm1: 1 - Opened or alarmed 0 - Closed or not alarned
1	Alarm2: 1 - Opened or alarmed

Table 64. 16ZoneStatus options*...continued*

Bit	Description
	0 - Closed or not alarned
2	Tamper: 1 - Tampered with 0 - Not tampered with
3	Battery: 1 - Low 0 - OK
4	Supervision reports[1]: 1 - Reports 0 - No reports
5	Restore reports[2]: 1 - Reports 0 - No reports
6	Trouble: 1 - Trouble/failure 0 - OK
7	AC (mains): 1 - Fault 0 - OK
8	Test mode: 1 - Sensor in test mode 0 - Sensor in operational mode
9	Battery defect: 1 - Defective battery detected 0 - Battery OK
10-15	Reserved

[1] Bit 4 indicates whether the Zone device issues periodic Zone Status Change Notification commands that may be used by the CIE device as evidence that the Zone device is operational.

[2] Bit 5 indicates whether the Zone device issues a Zone Status Change Notification command to notify when an alarm is no longer present (some Zone devices do not have the ability to detect when the alarm condition has disappeared).

‘Zone Settings’ Attribute Set

- `u64IASCIEAddress` is a mandatory attribute containing the 64-bit IEEE/MAC address of the CIE device to which the cluster server must send commands/ notifications
- `u8ZoneId` is a mandatory attribute containing the 8-bit identifier for the zone allocated by the CIE device at the time of enrollment
- `u8NumberOfZoneSensitivityLevels` is an optional attribute containing the number of sensitivity levels (for the detectable quantity) for the zone - for devices that have only one sensitivity level, this attribute need not be enabled or can be set to 0x00 or 0x01
  Note: *The definition of a sensitivity level is manufacturer-specific but detector ‘sensitivity’ should increase with higher values of this attribute.*
- `u16ClusterRevision` is a mandatory attribute that specifies the revision of the cluster specification on which this cluster instance is based. The cluster specification in the ZCL r6 corresponds to a cluster revision

of 1. The value is incremented by one for each subsequent revision of the cluster specification. This attribute is also described in Section 2.4

37.3 Enrollment

An IAS Zone device hosting the IAS Zone cluster server must be paired with a CIE device hosting the cluster client. This pairing is implemented by the process of 'enrollment' which, for extra security, provides a layer of pairing in addition to ZigBee PRO binding - if required, binding is implemented as part of the enrollment process.

During enrollment, the CIE device sends its IEEE/MAC address to the Zone device as well as a Zone ID, which is a unique 8-bit identifier that the CIE device assigns to the Zone device. These values are stored in the `u64IASCIEAddress` and `u8ZoneId` attributes on the Zone device (cluster server) - see Section 37.2. In addition, once enrollment has completed, the `e8ZoneState` attribute is set to 'enrolled'. Subsequently, the Zone device will only communicate with the paired CIE device.

Enrollment begins just after the Zone device joins the network. This device must then periodically poll for data (from the CIE device), ideally once every 2 seconds (or faster) but no slower than once every 7 seconds. This polling must continue until the `e8ZoneState` attribute has been updated to 'enrolled'. However, if the IAS Zone device supports the Poll Control cluster, polling at the above rate should continue until the Poll Control cluster configuration is changed.

Three methods of enrollment are available:

- Trip-to-Pair, described in Section 37.3.1
- Auto-Enroll-Response, described in Section 37.3.2
- Auto-Enroll-Request, described in Section 37.3.3

A cluster server and client can each implement both Trip-to-Pair and Auto-Enroll-Response or just Auto-Enroll-Request.

37.3.1 Trip-to-Pair

The Trip-to-Pair method of enrollment is described below:

1. After the IAS Zone device joins the network, the CIE device performs a service discovery.

2. If the CIE device determines that it wants to enroll the Zone device, it sends a Write Attribute command to the Zone device in order to write its IEEE/MAC address to the relevant attribute.

3. The Zone device may optionally create a binding table entry for the CIE device and store the CIE device's IEEE/MAC address there.

4. The Zone device waits for the authorization of the enrollment via a user input (for example, a button-press) and, on this input, sends a Zone Enroll Request command to the CIE device.

5. The CIE device assigns a Zone ID to the Zone device and sends a Zone Enroll Response command to it.

6. The Zone device updates its attributes to stored the assigned Zone ID and update its zone state to 'enrolled'.

37.3.2 Auto-Enroll-Response

The Auto-Enroll-Response method of enrollment is described below:

1. After the IAS Zone device joins the network, the CIE device performs a service discovery.

2. If the CIE device determines that it wants to enroll the Zone device, it sends a Write Attribute command to the Zone device in order to write its IEEE/MAC address to the relevant attribute.

3. The Zone device may optionally create a binding table entry for the CIE device and store the CIE device's IEEE/MAC address there.

4. The CIE device assigns a Zone ID to the Zone device and sends a Zone Enroll Response command to it.

5. The Zone device updates its attributes to stored the assigned Zone ID and update its zone state to 'enrolled'.

Note: *The above Auto-Enroll-Response process is similar to the Trip-to-Pair process (described in Section 37.3.2) except user authorization for the enrollment of the Zone device is not required and no Zone Enroll Request command needs to be sent to the CIE device.*

37.3.3 Auto-Enroll-Request

The Auto-Enroll-Request method of enrollment is described below:

1. After the IAS Zone device joins the network, the CIE device performs a service discovery.

2. If the CIE device determines that it wants to enroll the Zone device, it sends a `Write Attribute` command to the Zone device in order to write its IEEE/MAC address to the relevant attribute.

3. The Zone device may optionally create a binding table entry for the CIE device and store the CIE device's IEEE/MAC address there.

4. The Zone device sends a `Zone Enroll Request` command to the CIE device.

5. The CIE device assigns a `Zone ID` to the Zone device and sends a `Zone Enroll Response` command to it.

6. The Zone device updates its attributes to stored the assigned Zone ID and update its zone state to 'enrolled'.

Note: *The above Auto-Enroll-Request process is similar to the Trip-to-Pair process (described in Section 37.3.2) except that user authorization for the enrollment of the Zone device is not required.*

37.4 IAS Zone Events

The IAS Zone cluster has its own events that are handled through the callback mechanism outlined in Chapter 3. If a device uses the IAS Zone cluster then IAS Zone 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 **eHA_RegisterIASZoneEndPoint()** for a Zone device). The relevant callback function is then invoked when an IAS Zone event occurs.

For an IAS Zone 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 `tsCLD_IASZoneCallBackMessage` structure:

```
typedef struct
{
uint8 u8CommandId;
union
    {
     tsCLD_IASZone_TestModeUpdate                  *psTestModeUpdate; /*
 Internal */
     tsCLD_IASZone_EnrollRequestCallBackPayload
 sZoneEnrollRequestCallbackPayload;
     tsCLD_IASZone_EnrollResponsePayload
 *psZoneEnrollResponsePayload;
     tsCLD_IASZone_StatusChangeNotificationPayload
 *psZoneStatusNotificationPayload;
```

```
    tsCLD_IASZone_InitiateTestModeRequestPayload
 *psZoneInitiateTestModeRequestPayload;
   } uMessage;
} tsCLD_IASZoneCallBackMessage;
```

When an IAS Zone event occurs, one of several command types could have been received. The relevant command type is specified through the `u8CommandId` field of the `tsSM_CallBackMessage` structure. The possible command/event types are detailed in the table below (note that `psTestModeUpdate` is for internal use only).

Table 65. IAS Zone Command Types

u8CommandId Enumeration	Description
E_CLD_IASZONE_CMD_ZONE_ENROLL_RESP	An IAS Zone Enroll Response has been received by the cluster server
E_CLD_IASZONE_CMD_ZONE_STATUS_NOTIFICATION	An IAS Zone Status Change Notification has been received by the cluster client
E_CLD_IASZONE_CMD_ZONE_ENROLL_REQ	An IAS Zone Enroll Request has been received by the cluster client
E_CLD_IASZONE_CMD_INITIATE_NORMAL_OP_MODE_REQ	An IAS Zone Normal Operation Mode Initiation Request command has been received by the cluster server
E_CLD_IASZONE_CMD_INITIATE_TEST_MODE_REQ	An IAS Zone Initiate Test Mode Request has been received by the cluster server

37.5 Functions

The following IAS Zone cluster functions are provided in the NXP implementation of the ZCL:

1. eCLD_IASZoneCreateIASZone
2. eCLD_IASZoneUpdateZoneStatus
3. eCLD_IASZoneUpdateZoneState
4. eCLD_IASZoneUpdateZoneType
5. eCLD_IASZoneUpdateZoneID
6. eCLD_IASZoneUpdateCIEAddress
7. eCLD_IASZoneEnrollReqSend
8. eCLD_IASZoneEnrollRespSend
9. eCLD_IASZoneStatusChangeNotificationSend
10. eCLD_IASZoneNormalOperationModeReqSend
11. eCLD_IASZoneTestModeReqSend

37.5.1 eCLD_IASZoneCreateIASZone

```
teZCL_Status eCLD_IASZoneCreateIASZone(
tsZCL_ClusterInstance      *psClusterInstance,
bool_t     bIsServer,
tsZCL_ClusterDefinition     *psClusterDefinition,
void        *pvEndPointSharedStructPtr,
uint8         *pu8AttributeControlBits,
tsCLD_IASZone_CustomDataStructure *psCustomDataStructure);
```

Description

This function creates an instance of the IAS Zone 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 an IAS Zone cluster instance on the endpoint, but instances of other clusters may also be created on the same endpoint by calling their corresponding creation functions.

Note: *This function must not be called for an endpoint on which a standard ZigBee device is used. In this case, the device and its supported clusters must be registered on the endpoint using the relevant device registration function.*

When used, this function must be called after the stack has been started and after the ZCL has been initialized.

Parameters

- *psClusterInstance*: Pointer to structure containing information about the cluster instance to be created (see Section 6.1.16). This structure is updated by the function by initializing individual structure fields.
- *bIsServer* : Type of cluster instance (server or client) to be created: TRUE - serverFALSE - client
- *psClusterDefinition*: Pointer to structure indicating the type of cluster to be created (see Section 6.1.2). In this case, this structure must contain the details of the IAS Zone cluster. This parameter can refer to a pre-filled structure called `sCLD_IASZone` which is provided in the **IASZone.h** file.
- *pvEndPointSharedStructPtr*: Pointer to the shared structure used for attribute storage. This parameter should be the address of the structure of type `tsCLD_IASZone` which defines the attributes of IAS Zone cluster. The function initializes the attributes with default values.
- *pu8AttributeControlBits*: Pointer to an array of **uint8** values, with one element for each attribute in the cluster.
- *psCustomDataStructure*: Pointer to a structure containing the storage for internal functions of the cluster (see Section 37.6.1)

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_INVALID_VALUE

37.5.2 eCLD_IASZoneUpdateZoneStatus

```
teZCL_Status eCLD_IASZoneUpdateZoneStatus(
uint8       u8SourceEndPoint,
uint16      u16StatusBitMask,
bool_t      bStatusState);
```

Description

This function can be used on an IAS Zone cluster server to update the zone status bitmap stored in the `b16ZoneStatus` attribute, described in Section 37.2.

In one call to this function, one or more selected bits in the `b16ZoneStatus` attribute bitmap can be to set to ‘1’ or ‘0’. The affected bits must themselves be specified in a bitmap and the value to be set must also be specified.

If the server is enrolled with a client on a CIE device, the function sends a notification of this update to the client, in a Zone Status Change Notification. Before sending the notification and returning, the function invokes a user-defined callback function to allow the application to validate the status change.

Parameters

- *u8SourceEndPointId*: Number of the endpoint on which the IAS Zone cluster resides
- *u16StatusBitMask*: 16-bit bitmap indicating the bits of the `zb16ZoneStatus` bitmap to be updated. There is a one-to-one correspondence between the bits of the two bitmaps and a bit should be set to '1' if the corresponding attribute bit is to be updated. Enumerations are provided (which can be logical-ORed):

Table 66. *u16StatusBitMask* enumerations

Bits	Enumeration
0	CLD_IASZONE_STATUS_MASK_ALARM1
1	CLD_IASZONE_STATUS_MASK_ALARM2
2	CLD_IASZONE_STATUS_MASK_TAMPER
3	CLD_IASZONE_STATUS_MASK_BATTERY
4	CLD_IASZONE_STATUS_MASK_SUPERVISION_REPORTS
5	CLD_IASZONE_STATUS_MASK_RESTORE_REPORTS
6	CLD_IASZONE_STATUS_MASK_TROUBLE
7	CLD_IASZONE_STATUS_MASK_AC_MAINS
8	CLD_IASZONE_STATUS_MASK_TEST
9	CLD_IASZONE_STATUS_MASK_BATTERY_DEFECT
10-15	Reserved

- *bStatusState*: Boolean indicating the value to which the attribute bits to be updated must be set - enumerations are provided:
  - `CLD_IASZONE_STATUS_MASK_SET (1)`
  - `CLD_IASZONE_STATUS_MASK_RESET (0)`

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL

37.5.3 eCLD_IASZoneUpdateZoneState

```
teZCL_Status          eCLD_IASZoneUpdateZoneState(
uint8                u8SourceEndPoint,
 teCLD_IASZoneState   eZoneState);
```

Description

This function can be used on an IAS Zone cluster server to update the zone state value stored in the `e8ZoneState` attribute, described in Section 37.2. This attribute indicates whether or not the server is enrolled with a client on a CIE device. The function checks that the specified state is valid.

Parameters

- *u8SourceEndPointId*: Number of the endpoint on which the IAS Zone cluster resides
- *eZoneState*: Zone state value to be written to the attribute, one of:
  - E_CLD_IASZONE_STATE_NOT_ENROLLED (0x00)
  - E_CLD_IASZONE_STATE_ENROLLED (0x01)

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL

37.5.4 eCLD_IASZoneUpdateZoneType

```
teZCL_Status eCLD_IASZoneUpdateZoneType(
     uint8 u8SourceEndPoint,
 teCLD_IASZoneType eIASZoneType);
```

Description

This function can be used on an IAS Zone cluster server to update the zone type value stored in the `e16ZoneType` attribute. The possible values are listed in Section 37.2 and the function checks that the specified type is one of these values.

Parameters

- *u8SourceEndPointId*: Number of the endpoint on which the IAS Zone cluster resides *eIASZoneType*: Zone type value to be written to the attribute (for the possible values, refer to Section 37.2)

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL

37.5.5 eCLD_IASZoneUpdateZoneID

```
teZCL_Status eCLD_IASZoneUpdateZoneID(
        uint8 u8SourceEndPoint,
        uint8 u8IASZoneId);
```

Description

This function can be used on an IAS Zone cluster server to update the zone ID value stored in the `u8ZoneId` attribute. This is an 8-bit user-defined identifier.

Parameters

- *u8SourceEndPointId*: Number of the endpoint on which the IAS Zone cluster resides
- *u8IASZoneId*: Zone ID value to be written to the attribute

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL

37.5.6 eCLD_IASZoneUpdateCIEAddress

```
teZCL_Status eCLD_IASZoneUpdateCIEAddress(
        uint8 u8SourceEndPoint,
        u64IEEEAddress u64CIEAddress);
```

Description

This function can be used on an IAS Zone cluster server to update the 64-bit IEEE/MAC address stored in the `u64IASCIEAddress` attribute. This is the address of the CIE device to which the local device should send commands and notifications.

Parameters

- *u8SourceEndPointId*: Number of the endpoint on which the IAS Zone cluster resides
- *u64CIEAddress*: IEEE/MAC address to be written to the attribute

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL

37.5.7 eCLD_IASZoneEnrollReqSend

```
teZCL_Status eCLD_IASZoneEnrollReqSend(
        uint8 u8SourceEndPointId,
        uint8 u8DestinationEndPointId,
        tsZCL_Address *psDestinationAddress,
        uint8 *pu8TransactionSequenceNumber,
        tsCLD_IASZone_EnrollRequestPayload *psPayload);
```

Description

This function can be used on an IAS Zone cluster server to send an IAS Zone Enroll Request to an IAS Zone client.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is 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 to send the request. This parameter is used both to send the command and to identify the instance of the shared structure holding the required attribute values
- *u8DestinationEndPointId*: Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP

- *psDestinationAddress*: : Pointer to a structure holding the address of the node to which the request is sent
- *pu8TransactionSequenceNumber*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the request
- *psPayload*: Pointer to a structure containing the payload for the command (see Section 37.6.2)

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

37.5.8 eCLD_IASZoneEnrollRespSend

```
teZCL_Status eCLD_IASZoneEnrollRespSend(
        uint8 u8SourceEndPointId,
        uint8 u8DestinationEndPointId,
        tsZCL_Address *psDestinationAddress,
        uint8 *pu8TransactionSequenceNumber,
        tsCLD_IASZone_EnrollResponsePayload *psPayload);
```

Description

This function can be used on an IAS Zone cluster client to send an IAS Zone Enroll Response to the IAS Zone server.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is 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 to send the request. This parameter is used both to send the command and to identify the instance of the shared structure holding the required attribute values
- *u8DestinationEndPointId* Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP
- *psDestinationAddress* Pointer to a structure holding the address of the node to which the request is sent
- *pu8TransactionSequenceNumber* Pointer to a location to receive the Transaction Sequence Number (TSN) of the request
- *psPayload* Pointer to a structure containing the payload for the command (see Section 37.6.2)

Returns

- E_ZCL_SUCCESS

- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

37.5.9 eCLD_IASZoneStatusChangeNotificationSend

```
teZCL_Status eCLD_IASZoneStatusChangeNotificationSend(
        uint8 u8SourceEndPointId,
        uint8 u8DestinationEndPointId,
        tsZCL_Address *psDestinationAddress,
        uint8 *pu8TransactionSequenceNumber,
        tsCLD_IASZone_StatusChangeNotificationPayload
 *psPayload);
```

Description

This function can be used on IAS Zone cluster server to send a Zone Status Change Notification to the IAS Zone client.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is 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 to send the request. This parameter is used both to send the command and to identify the instance of the shared structure holding the required attribute values
- *u8DestinationEndPointId*: Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP
- *psDestinationAddress*: Pointer to a structure holding the address of the node to which the request is sent
- *pu8TransactionSequenceNumber*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the request
- *psPayload*: Pointer to a structure containing the payload for the command (see Section 37.6.2)

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

37.5.10 eCLD_IASZoneNormalOperationModeReqSend

```
teZCL_Status eCLD_IASZoneNormalOperationModeReqSend(
        uint8 u8SourceEndPointId,
        uint8 u8DestinationEndPointId,
        tsZCL_Address *psDestinationAddress,
        uint8 *pu8TransactionSequenceNumber);
```

Description

This function can be used on IAS Zone cluster client to send a request the IAS Zone server to initiate normal operation mode. If required, this command must be enabled in the compile-time options, as described in Section 37.7.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is 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 to send the request. This parameter is used both to send the command and to identify the instance of the shared structure holding the required attribute values
- *u8DestinationEndPointId*: Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP
- *psDestinationAddress*: Pointer to a structure holding the address of the node to which the request is sent
- *pu8TransactionSequenceNumber*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the request

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

37.5.11 eCLD_IASZoneTestModeReqSend

```
teZCL_Status eCLD_IASZoneTestModeReqSend(
    uint8 u8SourceEndPointId,
    uint8 u8DestinationEndPointId,
    tsZCL_Address *psDestinationAddress,
    uint8 *pu8TransactionSequenceNumber,
```

```
    tsCLD_IASZone_InitiateTestModeRequestPayload
                                                    *psPayload);
```

Description

This function can be used on IAS Zone cluster client to send a request to the IAS Zone server to initiate test mode and operate in this mode for a specified time. If required, this command must be enabled in the compile-time options, as described in Section 37.7.

Test mode allows the target device to be temporarily isolated from the IAS to allow configuration/adjustment of the device. Alternatively, the whole IAS can be put into test mode for maintenance, but the command issued by this function only affects the individual target IAS Zone cluster server(s).

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is 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 to send the request. This parameter is used both to send the command and to identify the instance of the shared structure holding the required attribute values
- *u8DestinationEndPointId*: Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP
- *psDestinationAddress*: Pointer to a structure holding the address of the node to which the request is sent
- *pu8TransactionSequenceNumber*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the request
- *psPayload*: Pointer to a structure containing the payload for the command (see Section 37.6.2)

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

37.6 Structures

37.6.1 Custom Data Structure

The IAS Zone cluster requires extra storage space to be allocated to be used by internal functions. The structure definition for this storage is shown below:

```
typedef struct
{
    tsCLD_IASZone_InitiateTestModeRequestPayload sTestMode;
    tsZCL_ReceiveEventAddress     sReceiveEventAddress;
```

```
    tsZCL_CallBackEvent           sCustomCallBackEvent;
    tsCLD_IASZoneCallBackMessage  sCallBackMessage;
} tsCLD_IASZone_CustomDataStructure;
```

The fields are for internal use and no knowledge of them is required.

37.6.2 Custom Command Payloads

The following structures contain the payloads for the IAS Zone cluster custom commands.

'Enroll Request' Payload

The following structure contains the payload of an Enroll Request command.

```
typedef struct
{
    zenum16    e16ZoneType;
    uint16     u16ManufacturerCode;
}tsCLD_IASZone_EnrollRequestPayload;
```

where:

- `e16ZoneType` is the zone type of the local (sending) node, as specified in the `e16ZoneType` attribute (see Section 37.2)
- `u16ManufacturerCode` is the manufacturer ID code that is held in the Node Descriptor of the local (sending) node

'Enroll Response' Payload

The following structure contains the payload of an Enroll Response command.

```
typedef struct
{
   teCLD_IASZoneZoneEnrollRspCode   e8EnrollResponseCode;
   uint8                            u8ZoneID;
}tsCLD_IASZone_EnrollResponsePayload;
```

where:

- `e8EnrollResponseCode` is a code indicating the outcome of the corresponding Enroll Request, one of:

Enumeration	Description
E_CLD_IASZONE_ENROLL_RESP_SUCCESS	Requested enrollment successful
E_CLD_IASZONE_ENROLL_RESP_NOT_SUPPORTED	Zone type of requesting device is not known/ sup-ported by the CIE device
E_CLD_IASZONE_ENROLL_RESP_NO_ENROLL_PERMIT	CIE device is not allowing new zones to be enrolled at the present time
E_CLD_IASZONE_ENROLL_RESP_TOO_MANY_ZONES	CIE device has reached its limit for the number of zones that it can enroll

- `u8ZoneID` is the index of the entry for the enrollment which has been added to the Zone table on the CIE device (only valid for a successful enrollment)

‘Zone Status Change Notification’ Payload

The following structure contains the payload of a Zone Status Change Notification command.

```
typedef struct
{
   zbmap16    b16ZoneStatus;
   zbmap8     b8ExtendedStatus;
   zuint8     u8ZoneId;
   zuint16    u16Delay;
}tsCLD_IASZone_StatusChangeNotificationPayload;
```

where:

- `b16ZoneStatus` contains the new/current status of the (sending) zone device, as indicated in the `e8ZoneState` attribute - one of:
  - E_CLD_IASZONE_STATE_NOT_ENROLLED (0x01)
  - E_CLD_IASZONE_STATE_ENROLLED (0x02)
- `b8ExtendedStatus` can be optionally used to indicate further status information, but otherwise should be set to zero
- `u8ZoneId` is the index of the entry for the (sending) device in the Zone table on the CIE device
- `u16Delay` is is the time-delay, in quarter-seconds, between the status change taking place in the `e8ZoneState` attribute and the successful transmission of the Zone Status Change Notification (this value can be used in assessing network traffic congestion)

‘Initiate Test Mode Request’ Payload

The following structure contains the payload of an Initiate Test Mode Request command.

```
typedef struct
{
    uint8    u8TestModeDuration;
    uint8    u8CurrentZoneSensitivityLevel;
}tsCLD_IASZone_InitiateTestModeRequestPayload;
```

where:

- `u8TestModeDuration` is the duration, in seconds, for which the device should remain in test mode
- `u8CurrentZoneSensitivityLevel` is the current sensitivity level for the zone, as indicated in the `u8CurrentZoneSensitivityLevel` attribute (see Section 37.2)

37.7 Compile-time options

To enable the IAS Zone cluster in the code to be built, it is necessary to add the following to the **zcl_options.h** file:

```
#define CLD_IASZONE
```

In addition, to include the software for a cluster client or server or both, it is necessary to add one of the following to the same file:

```
#define IASZONE_SERVER
#define IASZONE_CLIENT
```

Optional Attributes

Add this line to enable the optional Number Of Zone Sensitivity Levels attribute:

```
#define CLD_IASZONE_ATTR_ID_NUMBER_OF_ZONE_SENSITIVITY_LEVELS
```

Add this line to enable the optional Current Zone Sensitivity Level attribute:

```
#define CLD_IASZONE_ATTR_ID_CURRENT_ZONE_SENSITIVITY_LEVEL
```

Global Attributes

Add this line to define the value (n) of the Cluster Revision attribute:

```
#define CLD_IASZONE_CLUSTER_REVISION <n>
```

The default value is 1, which corresponds to the revision of the cluster in the ZCL r6 specification (see Section 2.4).

Optional Commands

Add this line to enable the optional Initiate Normal Operation Mode command:

```
#define CLD_IASZONE_CMD_INITIATE_NORMAL_OPERATION_MODE
```

Add this line to enable the optional Initiate Test Mode command:

```
#define CLD_IASZONE_CMD_INITIATE_TEST_MODE
```

Disable APS Acknowledgements for Bound Transmissions

APS acknowledgements for bound transmissions from this cluster can be disabled by defining:

```
#define CLD_IASZONE_BOUND_TX_WITH_APS_ACK_DISABLED
```

38 IAS Ancillary Control Equipment Cluster

This chapter describes the IAS Ancillary Control Equipment (ACE) cluster, which provides a control interface to a CIE (Control and Indicating Equipment) device in an IAS (Intruder Alarm System).

The IAS ACE cluster has a Cluster ID of 0x0501.

38.1 Overview

The IAS ACE cluster provides a control interface to a CIE (Control and Indicating Equipment) device in an IAS (Intruder Alarm System). For example, it allows a remote control unit to be used to configure the IAS via a CIE device. The server side of the cluster is implemented on the CIE device and the client side is implemented on the remote device.

To use the functionality of this cluster, you must include the file **IASACE.h** in your application and enable the cluster by defining CLD_IASACE in the **zcl_options.h** file.

The inclusion of the client or server software must be pre-defined in the application's compile-time options (in addition, if the cluster is to reside on a custom endpoint then the role of client or server must also be specified when creating the cluster instance).

The compile-time options for the IAS ACE cluster are fully detailed in Section 38.9.

38.2 IAS ACE Structure and Attributes

The structure definition for the IAS ACE cluster is shown below.

```
typedef struct
{
  zuint16     u16ClusterRevision;
}   tsCLD_IASACE;
```

where `u16ClusterRevision` is a mandatory attribute that specifies the revision of the cluster specification on which this cluster instance is based. The cluster specification in the ZCL r6 corresponds to a cluster revision of 1. The value is incremented by one for each subsequent revision of the cluster specification. This attribute is also described in Section 2.4.

38.3 Table and Parameters

The IAS ACE cluster server hosts the following table and sets of parameters:

- **Zone table:** The Zone table contains an entry for each enrolled zone. Each entry stores the identifier and type of the zone, as well as the IEEE/MAC address of the device which hosts the zone (see Section 38.7.2).
- **Zone parameters:** This set of parameters contains certain zone properties including the zone status, the zone name/label and the zone arm/disarm code (see Section 38.7.3)
- **Panel parameters:** This set of parameters contains certain status information about the display panel and alarm (see Section 38.7.4).

38.4 Command Summary

The IAS ACE cluster includes a number of commands that can be sent by the application on the client or server. These commands are summarised below.

- Table 50 lists the commands that can be issued on the client.
- Table 51 lists the commands that can be issued on the server.

Functions are provided to send these commands - these functions are indicated in the descriptions below and detailed in Section 38.6.

Table 67. IAS ACE Cluster Commands from Client to Server

Command	Description and Function
Arm	Instructs the server to put all or certain enrolled zones into the 'armed' state or put all of them into the 'disarmed' state. **eCLD_IASACE_ArmSend()**
Bypass	Instructs the server to take one or more specified zones out of the system for the current activation (these zones are reinstated the next time the system is dis-armed and to exclude them again the next time the system is armed, the Bypass command must be re-sent before sending the Arm command). **eCLD_IASACE_BypassSend()**
Emergency	Instructs the server to put the alarm in the 'Emergency' state. **eCLD_IASACE_EmergencySend()**
Fire	Instructs the server to put the alarm in the 'Fire' state. **eCLD_IASACE_FireSend()**
Panic	Instructs the server to put the alarm in the 'Panic' state. **eCLD_IASACE_PanicSend()**
Get Zone ID Map	Requests the Zone IDs that have been allocated to zones. **eCLD_IASACE_GetZoneIDMapSend()**
Get Zone Information	Requests information on a specified zone. **eCLD_IASACE_GetZoneInfoSend()**
Get Panel Status	Requests the current status of the (display) panel. **eCLD_IASACE_GetPanelStatusSend()**
Get Bypassed Zone List	Requests a list of the currently bypassed zones. **eCLD_IASACE_GetBypassedZoneListSend()**
Get Zone Status	Requests a list of either all zones with their status or those zones with a particular status (that is, all zones with the `b16ZoneStatus` attribute of the IAS Zone cluster having a certain value). **eCLD_IASACE_GetZoneStatusSend()**

Table 68. IAS ACE Cluster Commands from Server to Client

Command	Description and Function
Set Bypassed Zone List	Informs the client which zones are currently bypassed and can be sent in response to a Get Bypassed Zone List command. **eCLD_IASACE_SetBypassedZoneListSend()**
Zone Status Changed	Informs the client that the status (value of the `b16ZoneStatus` attribute of the IAS Zone cluster) of a particular zone has changed. **eCLD_IASACE_ZoneStatusChangedSend()**
Panel Status Changed	Informs the client that the status of the (display) panel has changed. **eCLD_IASACE_PanelStatusChanged()**

38.5 IAS ACE Events

The IAS ACE cluster has its own events that are handled through the callback mechanism outlined in Chapter 3. If a device uses the IAS ACE cluster then IAS ACE 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 **eHA_RegisterIASCIEEndPoint()** for a CIE device). The relevant callback function will then be invoked when an IAS ACE event occurs.

For an IAS ACE 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 `tsCLD_IASACECallBackMessage` structure:

```
typedef struct
{
uint8    u8CommandId;
  union
  {
  tsCLD_IASACE_ArmPayload  *psArmPayload;
  tsCLD_IASACE_BypassPayload     *psBypassPayload;
  tsCLD_IASACE_GetZoneInfoPayload   *psGetZoneInfoPayload;
  tsCLD_IASACE_GetZoneStatusPayload   *psGetZoneStatusPayload;
  tsCLD_IASACE_ArmRespPayload      *psArmRespPayload;
  tsCLD_IASACE_GetZoneIDMapRespPayload   *psGetZoneIDMapRespPayload;
  tsCLD_IASACE_GetZoneInfoRespPayload   *psGetZoneInfoRespPayload;
  tsCLD_IASACE_ZoneStatusChangedPayload  *psZoneStatusChangedPayload;
  tsCLD_IASACE_PanelStatusChangedOrGetPanelStatusRespPayload
   *psPanelStatusChangedOrGetPanelStatusRespPayload;
  tsCLD_IASACE_SetBypassedZoneListPayload    *psSetBypassedZoneListPayload;
  tsCLD_IASACE_BypassRespPayload    *psBypassRespPayload;
  tsCLD_IASACE_GetZoneStatusRespPayload   *psGetZoneStatusRespPayload;
    } uMessage;
} tsCLD_IASACECallBackMessage;
```

When an IAS ACE event occurs, one of twelve command types could have been received. The relevant command type is specified through the `u8CommandId` field of the `tsCLD_IASACECallBackMessage` structure. The possible command/event types are detailed in Table 52 below (for command descriptions, refer to Section 38.4).

In the case where an IAS Arm or Bypass command has been received and results in a change to a Zone parameter on the cluster server (e.g. an update of the zone status `u8ZoneStatusFlag`), a second event will be generated before any response is sent. This is a ‘cluster update’ event for which the `eEventType` field of the `tsZCL_CallBackEvent` structure is set to E_ZCL_CBET_CLUSTER_UPDATE. This prompts the application to perform any required actions such as saving persistent data and refreshing a display.

Table 69. IAS ACE Command Types

u8CommandId Enumeration	Description
Server Events	
E_CLD_IASACE_CMD_ARM	An IAS ACE Arm command has been received by the server
E_CLD_IASACE_CMD_BYPASS	An IAS ACE Bypass command has been received by the server
E_CLD_IASACE_CMD_EMERGENCY	An IAS ACE Emergency command has been received by the server
E_CLD_IASACE_CMD_FIRE	An IAS ACE Fire command has been received by the server
E_CLD_IASACE_CMD_PANIC	An IAS ACE Panic command has been received by the server
E_CLD_IASACE_CMD_GET_ZONE_ID_MAP	An IAS ACE Get Zone ID Map command has been received by the server
E_CLD_IASACE_CMD_GET_ZONE_INFO	An IAS ACE Get Zone Information command has been received by the server
E_CLD_IASACE_CMD_GET_PANEL_STATUS	An IAS ACE Get Panel Status command has been received by the server

Table 69. IAS ACE Command Types...*continued*

u8CommandId Enumeration	Description
E_CLD_IASACE_CMD_GET_BYPASSED_ZONE_LIST	An IAS ACE Get Bypassed Zone List command has been received by the server
E_CLD_IASACE_CMD_GET_ZONE_STATUS	An IAS ACE Get Zone Status command has been received by the server
Client Events	
E_CLD_IASACE_CMD_ARM_RESP	An IAS ACE Arm Response command has been received by the client
E_CLD_IASACE_CMD_GET_ZONE_ID_MAP_RESP	An IAS ACE Get Zone ID Map Response command has been received by the client
E_CLD_IASACE_CMD_GET_ZONE_INFO_RESP	An IAS ACE Get Zone Information Response command has been received by the client
E_CLD_IASACE_CMD_ZONE_STATUS_CHANGED	An IAS ACE Zone Status Changed command has been received by the client
E_CLD_IASACE_CMD_PANEL_STATUS_CHANGED	An IAS ACE Panel Status Changed command has been received by the client
E_CLD_IASACE_CMD_GET_PANEL_STATUS_RESP	An IAS ACE Get Panel Status Response command has been received by the client
E_CLD_IASACE_CMD_SET_BY-PASSED_ZONE_LIST	An IAS ACE Set Bypassed Zone List command has been received by the client
E_CLD_IASACE_CMD_BYPASS_RESP	An IAS ACE Bypass Response command has been received by the client
E_CLD_IASACE_CMD_GET_ZONE_STATUS_RESP	An IAS ACE Get Zone Status Response command has been received by the client

38.6 Functions

The following IAS ACE cluster functions are provided in the NXP implementation of the ZCL:

1. eCLD_IASACECreateIASACE
2. eCLD_IASACEAddZoneEntry
3. eCLD_IASACERemoveZoneEntry
4. eCLD_IASACEGetZoneTableEntry
5. eCLD_IASACEGetEnrolledZones
6. eCLD_IASACESetPanelParameter
7. eCLD_IASACEGetPanelParameter
8. eCLD_IASACESetZoneParameter
9. eCLD_IASACESetZoneParameterValue
10. eCLD_IASACEGetZoneParameter
11. eCLD_IASACE_ArmSend
12. eCLD_IASACE_BypassSend
13. eCLD_IASACE_EmergencySend
14. eCLD_IASACE_FireSend
15. eCLD_IASACE_PanicSend
16. eCLD_IASACE_GetZoneIDMapSend
17. eCLD_IASACE_GetZoneInfoSend

38.6.1 eCLD_IASACECreateIASACE

```
teZCL_Status eCLD_IASACECreateIASACE(
  tsZCL_ClusterInstance *psClusterInstance,
  bool_t bIsServer,
  tsZCL_ClusterDefinition *psClusterDefinition,
  void *pvEndPointSharedStructPtr,
  tsCLD_IASACECustomDataStructure
         *psCustomDataStructure);
```

Description

This function creates an instance of the IAS ACE 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 an IAS ACE cluster instance on the endpoint, but instances of other clusters may also be created on the same endpoint by calling their corresponding creation functions.

Note: *This function must not be called for an endpoint on which a standard ZigBee device is used. In this case, the device and its supported clusters must be registered on the endpoint using the relevant device registration function.*

When used, this function must be called after the stack has been started and after the ZCL has been initialized.

Parameters

- *psClusterInstance*: Pointer to structure containing information about the cluster instance to be created (see Section 6.1.16). This structure is updated by the function by initializing individual structure fields.
- *bIsServer*: Type of cluster instance (server or client) to be created: TRUE - server , FALSE - client
- *psClusterDefinition*: Pointer to structure indicating the type of cluster to be created (see Section 6.1.2). In this case, this structure must contain the details of the IAS ACE cluster.
- *pvEndPointSharedStructPtr*: Set this pointer to NULL for this cluster
- *psCustomDataStructure*: Pointer to a structure containing the storage for internal functions of the cluster (see Section 38.7.1)

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_INVALID_VALUE

38.6.2 eCLD_IASACEAddZoneEntry

```
teZCL_CommandStatus eCLD_IASACEAddZoneEntry(
  uint8 u8SourceEndPointId,
  uint16 u16ZoneType,
  uint64 u64IeeeAddress,
  uint8 *pu8ZoneID);
```

Description

This function can be used on an IAS ACE cluster server to create an entry in the local Zone table - that is, to add the details of a zone to the table after receiving a Zone Enrollment Request (and before sending a Zone Enrollment Response).

The details of the zone are provided in the function parameters. The function checks that the supplied pointer to the Zone ID is not NULL and that the supplied IEEE address is valid. The function can then add the zone details to the Zone table, provided that there is a free entry in the table.

Parameters

- *u8SourceEndPointId*: Number of the endpoint on which the IAS ACE cluster resides
- *u16ZoneType*: Value indicating the type of zone to be added to the table (for the possible values, refer to the description of the attribute `e16ZoneType` of the IAS Zone cluster in Section 37.2)
- *u64IeeeAddress*: IEEE address of the device which hosts the zone
- *pu8ZoneID*: Pointer to an identifier of the zone to be added to the table

Returns

- E_ZCL_CMDS_SUCCESS (zone successfully added to Zone table)
- E_ZCL_CMDS_FAILURE (cluster instance not found)
- E_ZCL_CMDS_INVALID_FIELD (pointer to Zone ID is NULL)
- E_ZCL_CMDS_INVALID_VALUE (IEEE address is invalid)
- E_ZCL_CMDS_INSUFFICIENT_SPACE (no free entry in Zone table)

38.6.3 eCLD_IASACERemoveZoneEntry

```
teZCL_CommandStatus eCLD_IASACERemoveZoneEntry(
  uint8 u8SourceEndPointId,
  uint8 u8ZoneID,
  uint64 *pu64IeeeAddress);
```

Description

This function can be used on an IAS ACE cluster server to remove an existing entry from the local Zone table - that is, to delete the details of a zone in the table and release the table entry for re-use. Thus, this function can be used to unenroll a zone.

The zone to be removed is specified by means of the Zone ID. The function checks that the supplied pointer to a location to receive the IEEE address is not NULL. The function then searches for the relevant table entry using the supplied Zone ID and, if found, returns its IEEE address via the supplied location and frees the table entry by setting the IEEE address in the table entry to zero. The returned IEEE address can be used by a

(local) CIE device application to send a request to the relevant Zone device to set its IAS Zone cluster attribute `u64IASCIEAddress` to all zeros (writing to remote attributes is described in Section 2.3.3.1).

Parameters

- *u8SourceEndPointId*: Number of the endpoint on which the IAS ACE cluster resides
- *u8ZoneID*: Zone ID of zone to be removed from table
- *pu64IeeeAddress*: Pointer to location to receive the IEEE address found in the table entry to be removed

Returns

- E_ZCL_CMDS_SUCCESS (zone successfully removed from Zone table)
- E_ZCL_CMDS_FAILURE (cluster instance not found)
- E_ZCL_CMDS_INVALID_FIELD (pointer to IEEE address location is NULL)
- E_ZCL_CMDS_NOT_FOUND (entry with specified Zone ID not found in table)

38.6.4 eCLD_IASACEGetZoneTableEntry

```
teZCL_CommandStatus eCLD_IASACEGetZoneTableEntry(
  uint8 u8SourceEndPointId,
  uint8 u8ZoneID,
  tsCLD_IASACE_ZoneTable **ppsZoneTable);
```

Description

This function can be used on an IAS ACE cluster server to obtain the details of a specified zone from the local Zone table.

The zone of interest is specified by means of its Zone ID. The function searches for the relevant table entry using the supplied Zone ID and, if found, returns the zone information from the table entry via the supplied structure (see Section 38.7.2).

Parameters

- *u8SourceEndPointId*: Number of the endpoint on which the IAS ACE cluster resides
- *u8ZoneID*: Zone ID of zone for which details required from table
- *ppsZoneTable*: Pointer to a pointer to a structure to receive obtained zone information (see Section 38.7.2)

Returns

- E_ZCL_CMDS_SUCCESS (zone details successfully obtained from Zone table)
- E_ZCL_CMDS_FAILURE (cluster instance not found)
- E_ZCL_CMDS_NOT_FOUND (entry with specified Zone ID not found in table)

38.6.5 eCLD_IASACEGetEnrolledZones

```
teZCL_CommandStatus eCLD_IASACEGetEnrolledZones(
  uint8 u8SourceEndPointId,
  uint8 *pu8ZoneID,
  uint8 *pu8NumOfEnrolledZones);
```

Description

This function can be used on an IAS ACE cluster server to obtain a list of the enrolled zones from the local Zone table.

The function searches the Zone table and returns a list of the Zone IDs of all the enrolled zones (for which there are table entries). The number of enrolled zones is also returned.

Parameters

- *u8SourceEndPointId*: Number of the endpoint on which the IAS ACE cluster resides
- *pu8ZoneID*: Pointer to a location to receive the first Zone ID in the reported list of enrolled zones
- *pu8NumOfEnrolledZones*: Pointer to a location to receive the number of enrolled zones reported in the above list

Returns

- E_ZCL_CMDS_SUCCESS (zone list successfully obtained from Zone table)
- E_ZCL_CMDS_FAILURE (cluster instance not found)
- E_ZCL_CMDS_INVALID_FIELD (a supplied pointer is NULL)

38.6.6 eCLD_IASACESetPanelParameter

```
teZCL_Status eCLD_IASACESetPanelParameter(
  uint8 u8SourceEndPointId,
  teCLD_IASACE_PanelParameterID eParameterId,
  uint8 u8ParameterValue);
```

Description

This function can be used on an IAS ACE cluster server to set the value of a Panel parameter. The Panel parameters are held on the server in a `tsCLD_IASACE_PanelParameter` structure (see Section 38.7.4) and this function can be used to write a value to one parameter in the structure. The function verifies that the specified parameter identifier is valid before attempting the write.

If this function is used to set the Panel parameter `ePanelStatus`, an IAS ACE Panel Status Changed command is automatically sent to all bound clients.

Parameters

- *u8SourceEndPointId*: Number of the endpoint on which the IAS ACE cluster resides
- *eParameterId*: Enumeration identifying the Panel parameter to be set, one of:
  - E_CLD_IASACE_PANEL_PARAMETER_PANEL_STATUS
  - E_CLD_IASACE_PANEL_PARAMETER_SECONDS_REMAINING
  - E_CLD_IASACE_PANEL_PARAMETER_AUDIBLE_NOTIFICATION
  - E_CLD_IASACE_PANEL_PARAMETER_ALARM_STATUS
- *u8ParameterValue*: Value to be written to the parameter

Returns

- E_ZCL_SUCCESS (Panel parameter successfully set)
- E_ZCL_ERR_CLUSTER_NOT_FOUND (cluster instance not found)

- E_ZCL_ERR_ATTRIBUTE_NOT_FOUND (Panel parameter identifier invalid)

38.6.7 eCLD_IASACEGetPanelParameter

```
teZCL_Status eCLD_IASACEGetPanelParameter(
  uint8 u8SourceEndPointId,
  teCLD_IASACE_PanelParameterID eParameterId,
  uint8 *pu8ParameterValue);
```

Description

This function can be used on an IAS ACE cluster server to obtain the value of a Panel parameter. The Panel parameters are held on the server in a `tsCLD_IASACE_PanelParameter` structure (see Section 38.7.4) and this function can be used to read the value of one parameter in the structure. The function verifies that the specified parameter identifier is valid before attempting the read.

Parameters

- *u8SourceEndPointId*: Number of the endpoint on which the IAS ACE cluster resides
- *eParameterId* : Enumeration identifying the Panel parameter to be read, one of:
- E_CLD_IASACE_PANEL_PARAMETER_PANEL_STATUS
- E_CLD_IASACE_PANEL_PARAMETER_SECONDS_REMAINING
- E_CLD_IASACE_PANEL_PARAMETER_AUDIBLE_NOTIFICATION
- E_CLD_IASACE_PANEL_PARAMETER_ALARM_STATUS
- *pu8ParameterValue*: Pointer to location to receive read parameter value

Returns

- E_ZCL_SUCCESS (Panel parameter successfully read)
- E_ZCL_ERR_CLUSTER_NOT_FOUND (cluster instance not found)
- E_ZCL_ERR_PARAMETER_NULL (specfied pointer is NULL)
- E_ZCL_ERR_ATTRIBUTE_NOT_FOUND (Panel parameter identifier invalid)

38.6.8 eCLD_IASACESetZoneParameter

```
teZCL_Status eCLD_IASACESetZoneParameter(
  uint8 u8SourceEndPointId,
  teCLD_IASACE_ZoneParameterID eParameterId,
  uint8 u8ZoneID,
  uint8 u8ParameterLength,
  uint8 *pu8ParameterValue);
```

Description

This function can be used on an IAS ACE cluster server to set the value of a Zone parameter. The Zone parameters for a particular Zone ID are held on the server in a `tsCLD_IASACE_ZoneParameter` structure (see Section 38.7.3) and this function can be used to write a value to one parameter in the structure. The specified zone must have been enrolled in the local Zone table. Before attempting the write, the function verifies that the specified Zone ID is present in the Zone table and that the specified parameter identifier is valid.

If this function is used to set the Zone parameter `eZoneStatus`, an IAS ACE Zone Status Changed command is automatically sent to all bound clients.

The function requires the parameter value to be provided as a **uint8** array. This is to allow one of the array parameters, `au8ZoneLabel[]` or `au8ArmDisarmCode[]`, to be set - the corresponding string parameter, `sZoneLabel` or `sArmDisarmCode`, will be set automatically. The function **eCLD_IASACESetZoneParameterValue()** provides an easier way of setting one of the non-array/non-string parameters.

Parameters

- *u8SourceEndPointId*: Number of the endpoint on which the IAS ACE cluster resides
- *eParameterId* : Enumeration identifying the Zone parameter to be set, one of:
- : E_CLD_IASACE_ZONE_PARAMETER_ZONE_CONFIG_FLAG
- E_CLD_IASACE_ZONE_PARAMETER_ZONE_STATUS_FLAG
- E_CLD_IASACE_ZONE_PARAMETER_ZONE_STATUS
- E_CLD_IASACE_ZONE_PARAMETER_AUDIBLE_NOTIFICATION
- E_CLD_IASACE_ZONE_PARAMETER_ZONE_LABEL
- E_CLD_IASACE_ZONE_PARAMETER_ARM_DISARM_CODE
- *u8ZoneID*: Zone ID of zone information to be updated
- *u8ParameterLength*: Number of **uint8** elements in the array containing the parameter value to be set
- *pu8ParameterValue*: Pointer to a location containing the first element of the array containing the parameter value to be set

Returns

- E_ZCL_SUCCESS (Zone parameter successfully set)
- E_ZCL_ERR_CLUSTER_NOT_FOUND (cluster instance not found)
- E_ZCL_ERR_ATTRIBUTE_NOT_FOUND (Zone parameter identifier invalid)
- E_ZCL_ERR_NO_REPORT_ENTRIES (Zone ID not found in Zone table)
- E_ZCL_ERR_PARAMETER_NULL (Pointer to location containing value is NULL)
- E_ZCL_ERR_PARAMETER_RANGE (specified array length too long to be stored)

38.6.9 eCLD_IASACESetZoneParameterValue

```
teZCL_Status eCLD_IASACESetZoneParameterValue(
  uint8 u8SourceEndPointId,
  teCLD_IASACE_ZoneParameterID eParameterId,
  uint8 u8ZoneID,
  uint16 u16ParameterValue);
```

Description

This function can be used on an IAS ACE cluster server to set the value of a Zone parameter. The Zone parameters for a particular Zone ID are held on the server in a `tsCLD_IASACE_ZoneParameter` structure (see Section 38.7.3) and this function can be used to write a value to one of the non-string/non-array parameters in the structure. The specified zone must have been enrolled in the local Zone table. Before attempting the write, the function verifies that the specified Zone ID is present in the Zone table and that the specified parameter identifier is valid.

If this function is used to set the Zone parameter `eZoneStatus`, an IAS ACE Zone Status Changed command is automatically sent to all bound clients.

This function cannot be used to set the string parameters `sZoneLabel` and `sArmDisarmCode` or the array parameters `au8ZoneLabel[]` and `au8ArmDisarmCode[]`. The function **eCLD_IASACESetZoneParameter()** must be used to set the string and array parameters.

Parameters

- *u8SourceEndPointId*: Number of the endpoint on which the IAS ACE cluster resides
- *eParameterId* : Enumeration identifying the Zone parameter to be set, one of:
  - E_CLD_IASACE_ZONE_PARAMETER_ZONE_CONFIG_FLAG
  - E_CLD_IASACE_ZONE_PARAMETER_ZONE_STATUS_FLAG
  - E_CLD_IASACE_ZONE_PARAMETER_ZONE_STATUS
  - E_CLD_IASACE_ZONE_PARAMETER_AUDIBLE_NOTIFICATION
- *u8ZoneID*: Zone ID of zone information to be updated
- *u16ParameterValue*: Value to be written to the parameter

Returns

- E_ZCL_SUCCESS (Zone parameter successfully set)
- E_ZCL_ERR_CLUSTER_NOT_FOUND (cluster instance not found)
- E_ZCL_ERR_ATTRIBUTE_NOT_FOUND (Zone parameter identifier invalid)
- E_ZCL_ERR_NO_REPORT_ENTRIES (Zone ID not found in Zone table)

38.6.10 eCLD_IASACEGetZoneParameter

```
teZCL_Status eCLD_IASACEGetZoneParameter(
  uint8 u8SourceEndPointId,
  teCLD_IASACE_ZoneParameterID eParameterId,
  uint8 u8ZoneID,
  uint8 *pu8ParameterLength,
  uint8 *pu8ParameterValue);
```

Description

This function can be used on an IAS ACE cluster server to obtain the value of a Zone parameter. The Zone parameters for a particular Zone ID are held on the server in a `tsCLD_IASACE_ZoneParameter` structure (see Section 38.7.3) and this function can be used to read the value of one parameter in the structure. Before attempting the read, the function verifies that the specified Zone ID is present in the Zone table and that the specified parameter identifier is valid.

The function expects the read parameter value to be returned as a **uint8** array.

Parameters

- *u8SourceEndPointId*: Number of the endpoint on which the IAS ACE cluster resides
- *eParameterId*: Enumeration identifying the Zone parameter to be read, one of:
  - E_CLD_IASACE_ZONE_PARAMETER_ZONE_CONFIG_FLAG
  - E_CLD_IASACE_ZONE_PARAMETER_ZONE_STATUS_FLAG
  - E_CLD_IASACE_ZONE_PARAMETER_ZONE_STATUS
  - E_CLD_IASACE_ZONE_PARAMETER_AUDIBLE_NOTIFICATION
  - E_CLD_IASACE_ZONE_PARAMETER_ZONE_LABEL
  - E_CLD_IASACE_ZONE_PARAMETER_ARM_DISARM_CODE

- *u8ZoneID*: Zone ID of zone information to be accessed
- *pu8ParameterLength*: Pointer to location to receive the number of **uint8** elements in the array containing the parameter value obtained
- *pu8ParameterValue*: Pointer to location to receive the first element of the array containing the parameter value obtained

Returns

- E_ZCL_SUCCESS (Zone parameter successfully read)
- E_ZCL_ERR_CLUSTER_NOT_FOUND (cluster instance not found)
- E_ZCL_ERR_PARAMETER_NULL (a specified pointer is NULL)
- E_ZCL_ERR_NO_REPORT_ENTRIES (Zone ID not found in Zone table)
- E_ZCL_ERR_ATTRIBUTE_NOT_FOUND (Zone parameter identifier invalid)
- E_ZCL_ERR_PARAMETER_RANGE (returned array too long to be stored)

38.6.11 eCLD_IASACE_ArmSend

```
teZCL_Status eCLD_IASACE_ArmSend(
  uint8 u8SourceEndPointId,
  uint8 u8DestinationEndPointId,
  tsZCL_Address *psDestinationAddress,
  uint8 *pu8TransactionSequenceNumber,
  tsCLD_IASACE_ArmPayload *psPayload);
```

Description

This function can be used on an IAS ACE cluster client to send an IAS ACE Arm command to an IAS ACE server. This command instructs the server to put all or certain enrolled zones into the ‘armed’ state or put all of them into the ‘disarmed’ state, according to the command payload (see Section 38.7.5).

The outcome of the request will be returned by the server in a response which will generate an E_CLD_IASACE_CMD_ARM_RESP event when received on the client.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is 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 to send the request. This parameter is used both to send the command and to identify the instance of the shared structure holding the required attribute values
- *u8DestinationEndPointId*: Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP
- *psDestinationAddress*: Pointer to a structure holding the address of the node to which the request is sent
- *pu8TransactionSequenceNumber*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the request
- *psPayload*: Pointer to a structure containing the payload for the command (see Section 38.7.5)

Returns

- E_ZCL_SUCCESS

- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

38.6.12 eCLD_IASACE_BypassSend

```
teZCL_Status eCLD_IASACE_BypassSend(
  uint8 u8SourceEndPointId,
  uint8 u8DestinationEndPointId,
  tsZCL_Address *psDestinationAddress,
  uint8 *pu8TransactionSequenceNumber,
  tsCLD_IASACE_BypassPayload *psPayload);
```

Description

This function can be used on an IAS ACE cluster client to send an IAS ACE Bypass command to an IAS ACE server. This command instructs the server to take one or more specified zones out of the system for the current activation.

Note: *The bypassed zones will be reinstated the next time the system is disarmed. To exclude them again the next time the system is armed, the Bypass command must be re-sent before sending the Arm command.*

The outcome of the request will be returned by the server in a response which will generate an E_CLD_IASACE_CMD_BYPASS_RESP event when received on the client.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is 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 to send the request. This parameter is used both to send the command and to identify the instance of the shared structure holding the required attribute values
- *u8DestinationEndPointId*: Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP
- *psDestinationAddress*: Pointer to a structure holding the address of the node to which the request is sent
- *pu8TransactionSequenceNumber*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the request
- *psPayload*: Pointer to a structure containing the payload for the command (see Section 38.7.5)

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN

- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

38.6.13 eCLD_IASACE_EmergencySend

```
teZCL_Status eCLD_IASACE_EmergencySend(
  uint8 u8SourceEndPointId,
  uint8 u8DestinationEndPointId,
  tsZCL_Address *psDestinationAddress,
  uint8 *pu8TransactionSequenceNumber);
```

Description

This function can be used on an IAS ACE cluster client to send an IAS ACE Emergency command to an IAS ACE server. This command instructs the server to put the alarm in the ‘Emergency’ state.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is 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 to send the request. This parameter is used both to send the command and to identify the instance of the shared structure holding the required attribute values
- *u8DestinationEndPointId*: Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP
- *psDestinationAddress*: Pointer to a structure holding the address of the node to which the request is sent
- *pu8TransactionSequenceNumber*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the request

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

38.6.14 eCLD_IASACE_FireSend

```
teZCL_Status eCLD_IASACE_FireSend(
  uint8 u8SourceEndPointId,
```

```
  uint8 u8DestinationEndPointId,
  tsZCL_Address *psDestinationAddress,
  uint8 *pu8TransactionSequenceNumber);
```

Description

This function can be used on an IAS ACE cluster client to send an IAS ACE Fire command to an IAS ACE server. This command instructs the server to put the alarm in the ‘Fire’ state.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is 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 to send the request. This parameter is used both to send the command and to identify the instance of the shared structure holding the required attribute values
- *u8DestinationEndPointId*: Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP
- *psDestinationAddress*: Pointer to a structure holding the address of the node to which the request is sent
- *pu8TransactionSequenceNumber*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the request

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

38.6.15 eCLD_IASACE_PanicSend

```
teZCL_Status eCLD_IASACE_PanicSend(
  uint8 u8SourceEndPointId,
  uint8 u8DestinationEndPointId,
  tsZCL_Address *psDestinationAddress,
  uint8 *pu8TransactionSequenceNumber);
```

Description

This function can be used on an IAS ACE cluster client to send an IAS ACE Panic command to an IAS ACE server. This command instructs the server to put the alarm in the ‘Panic’ state.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is 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 to send the request. This parameter is used both to send the command and to identify the instance of the shared structure holding the required attribute values
- *u8DestinationEndPointId*: Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP
- *psDestinationAddress*: Pointer to a structure holding the address of the node to which the request is sent
- *pu8TransactionSequenceNumber*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the request

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

38.6.16 eCLD_IASACE_GetZoneIDMapSend

```
teZCL_Status eCLD_IASACE_GetZoneIDMapSend(
  uint8 u8SourceEndPointId,
  uint8 u8DestinationEndPointId,
  tsZCL_Address *psDestinationAddress,
  uint8 *pu8TransactionSequenceNumber);
```

Description

This function can be used on an IAS ACE cluster client to send an IAS ACE Get Zone ID Map command to an IAS ACE server. This command requests the Zone IDs that have been allocated to zones.

The requested information is returned by the server in a response which generates an E_CLD_IASACE_CMD_GET_ZONE_ID_MAP_RESP event when received on the client.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is 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 to send the request. This parameter is used both to send the command and to identify the instance of the shared structure holding the required attribute values
- *u8DestinationEndPointId*: Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP
- *psDestinationAddress*: Pointer to a structure holding the address of the node to which the request is sent

- *pu8TransactionSequenceNumber*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the request

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

38.6.17 eCLD_IASACE_GetZoneInfoSend

```
teZCL_Status eCLD_IASACE_GetZoneInfoSend(
  uint8 u8SourceEndPointId,
  uint8 u8DestinationEndPointId,
  tsZCL_Address *psDestinationAddress,
  uint8 *pu8TransactionSequenceNumber,
  tsCLD_IASACE_GetZoneInfoPayload *psPayload);
```

Description

This function can be used on an IAS ACE cluster client to send an IAS ACE Get Zone Information command to an IAS ACE server. This command requests information on the zone specified in the command payload.

The requested information will be returned by the server in a response which will generate an E_CLD_IASACE_CMD_GET_ZONE_INFO_RESP event when received on the client.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is 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 to send the request. This parameter is used both to send the command and to identify the instance of the shared structure holding the required attribute values
- *u8DestinationEndPointId*: Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP
- *psDestinationAddress*: Pointer to a structure holding the address of the node to which the request is sent
- *pu8TransactionSequenceNumber*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the request
- *psPayload*: Pointer to a structure containing the payload for the command (see Section 38.7.5)

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL

- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

38.6.18 eCLD_IASACE_GetPanelStatusSend

```
teZCL_Status eCLD_IASACE_GetPanelStatusSend(
  uint8 u8SourceEndPointId,
  uint8 u8DestinationEndPointId,
  tsZCL_Address *psDestinationAddress,
  uint8 *pu8TransactionSequenceNumber);
```

Description

This function can be used on an IAS ACE cluster client to send an IAS ACE Get Panel Status command to an IAS ACE server. This command requests the current status of the (display) panel.

The requested information will be returned by the server in a response which will generate an E_CLD_IASACE_CMD_GET_PANEL_STATUS_RESP event when received on the client.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is 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 to send the request. This parameter is used both to send the command and to identify the instance of the shared structure holding the required attribute values
- *u8DestinationEndPointId*: Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP
- *psDestinationAddress*: Pointer to a structure holding the address of the node to which the request is sent
- *pu8TransactionSequenceNumber*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the request

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

38.6.19 eCLD_IASACE_SetBypassedZoneListSend

```
teZCL_Status eCLD_IASACE_SetBypassedZoneListSend(
  uint8 u8SourceEndPointId,
  uint8 u8DestinationEndPointId,
  tsZCL_Address *psDestinationAddress,
  uint8 *pu8TransactionSequenceNumber,
  tsCLD_IASACE_SetBypassedZonelistPayload *psPayload);
```

Description

This function can be used on an IAS ACE cluster server to send an IAS ACE Set Bypassed Zone List command to an IAS ACE client. This command informs the client which zones are currently bypassed - the zones are specified in the command payload.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is 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 to send the request. This parameter is used both to send the command and to identify the instance of the shared structure holding the required attribute values
- *u8DestinationEndPointId*: Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP
- *psDestinationAddress*: Pointer to a structure holding the address of the node to which the request is sent
- *pu8TransactionSequenceNumber*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the request
- *psPayload*: Pointer to a structure containing the payload for the command (see Section 38.7.5)

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

38.6.20 eCLD_IASACE_GetBypassedZoneListSend

```
teZCL_Status eCLD_IASACE_GetBypassedZoneListSend(
  uint8 u8SourceEndPointId,
  uint8 u8DestinationEndPointId,
  tsZCL_Address *psDestinationAddress,
  uint8 *pu8TransactionSequenceNumber);
```

Description

This function can be used on an IAS ACE cluster client to send an IAS ACE Get Bypassed Zone List command to an IAS ACE server. This command requests a list of the currently bypassed zones.

The requested information will be returned by the server in a response which will generate an E_CLD_IASACE_CMD_SET_BYPASSED_ZONE_LIST event when received on the client.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is 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 to send the request. This parameter is used both to send the command and to identify the instance of the shared structure holding the required attribute values
- *u8DestinationEndPointId*: Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP
- *psDestinationAddress*: Pointer to a structure holding the address of the node to which the request is sent
- *pu8TransactionSequenceNumber*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the request

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

38.6.21 eCLD_IASACE_GetZoneStatusSend

```
teZCL_Status eCLD_IASACE_GetZoneStatusSend(
  uint8 u8SourceEndPointId,
  uint8 u8DestinationEndPointId,
  tsZCL_Address *psDestinationAddress,
  uint8 *pu8TransactionSequenceNumber,
  tsCLD_IASACE_GetZoneStatusPayload *psPayload);
```

Description

This function can be used on an IAS ACE cluster client to send an IAS ACE Get Zone Status command to an IAS ACE server. This command requests either of the following:

- a list of all enrolled zones with their status
- a list of those zones with a particular status (that is, all zones with the `b16ZoneStatus` attribute of the IAS Zone cluster having a certain value)

The list required is specified in the `bZoneStatusMaskFlag` field of the command payload (see Section 38.7.5). If the second of the above lists is required, the status to look for is also specified in the payload.

The requested information is returned by the server in a response which generates an E_CLD_IASACE_CMD_GET_ZONE_STATUS_RESP event when received on the client. A single response may not be able to carry all the zone status information to be returned and more than one request (and associated response) would be needed. For this reason, the request allows a starting zone and the number of zones to be included in the response to be specified (in the request payload).

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is 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 to send the request. This parameter is used both to send the command and to identify the instance of the shared structure holding the required attribute values
- *u8DestinationEndPointId*: Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP
- *psDestinationAddress*: Pointer to a structure holding the address of the node to which the request is sent
- *pu8TransactionSequenceNumber*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the request
- *psPayload*: Pointer to a structure containing the payload for the command (see Section 38.7.5)

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

38.6.22 eCLD_IASACE_ZoneStatusChangedSend

```
teZCL_Status eCLD_IASACE_ZoneStatusChangedSend(
  uint8 u8SourceEndPointId,
  uint8 u8DestinationEndPointId,
  tsZCL_Address *psDestinationAddress,
  uint8 *pu8TransactionSequenceNumber,
  tsCLD_IASACE_ZoneStatusChangedPayload *psPayload);
```

Description

This function can be used on an IAS ACE cluster server to send an IAS ACE Zone Status Changed command to an IAS ACE client. This command informs the client that the status of the specified zone has changed - that is, the value of the `b16ZoneStatus` attribute of the IAS Zone cluster for the zone has changed.

Note: *This command is sent automatically when the function* ***eCLD_IASACESetZoneParameter()*** *is called on the server to update the* `u16ZoneStatus` *attribute for all the bound clients.*

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is 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 to send the request. This parameter is used both to send the command and to identify the instance of the shared structure holding the required attribute values
- *u8DestinationEndPointId*: Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP
- *psDestinationAddress*: Pointer to a structure holding the address of the node to which the request is sent
- *pu8TransactionSequenceNumber*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the request
- *psPayload*: Pointer to a structure containing the payload for the command (see Section 38.7.5)

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

38.6.23 eCLD_IASACE_PanelStatusChanged

```
teZCL_Status eCLD_IASACE_PanelStatusChanged(
  uint8 u8SourceEndPointId,
  uint8 u8DestinationEndPointId,
  tsZCL_Address *psDestinationAddress,
  uint8 *pu8TransactionSequenceNumber,
   teCLD_IASACE_ServerCmdId      eCommandId,
 tsCLD_IASACE_PanelStatusChangedOrGetPanelStatusRespPayload
        *psPayload);
```

Description

This function can be used on an IAS ACE cluster server to send an IAS ACE Panel Status Changed command to an IAS ACE client. This command informs the client that the value of the panel parameter `ePanelStatus` (see Section 38.7.4) on the (local) CIE device has changed.

Note:

1. *The IAS ACE Panel Status Changed command is sent automatically when the function* ***eCLD_IASACESetPanelParameter()*** *is called to update the* `ePanelStatus` *parameter.*

2. *The function alternatively provides the option of sending an IAS ACE Get Panel Status Response but, in practice, this response is sent automatically when a Get Panel Status Request is received.*

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is 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 to send the request. This parameter is used both to send the command and to identify the instance of the shared structure holding the required attribute values
- *u8DestinationEndPointId*: Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP
- *psDestinationAddress*: Pointer to a structure holding the address of the node to which the request is sent
- *pu8TransactionSequenceNumber*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the request
- *eCommandId*: Identifier of command to be sent - for Panel Status Changed command, always set to:: E_CLD_IASACE_CMD_PANEL_STATUS_CHANGED
- *psPayload*: Pointer to a structure containing the payload for the command (see Section 38.7.5)

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

38.7 Structures

38.7.1 Custom Data Structure

The IAS ACE cluster requires extra storage space to be allocated to be used by internal functions. The structure definition for this storage is shown below:

```
typedef struct
{
    tsZCL_ReceiveEventAddress        sReceiveEventAddress;
    tsZCL_CallBackEvent              sCustomCallBackEvent;
    tsCLD_IASACECallBackMessage      sCallBackMessage;
#if (defined CLD_IASACE) && (defined IASACE_SERVER)
    tsCLD_IASACE_PanelParameter
                                     sCLD_IASACE_PanelParameter;
    tsCLD_IASACE_ZoneParameter
           asCLD_IASACE_ZoneParameter[CLD_IASACE_ZONE_TABLE_SIZE];
    tsCLD_IASACE_ZoneTable
           asCLD_IASACE_ZoneTable[CLD_IASACE_ZONE_TABLE_SIZE];
#endif
```

```
} tsCLD_IASACECustomDataStructure;
```

The fields are for internal use and no knowledge of them is required.

38.7.2 Zone Table Entry

The following structure contains a Zone table entry, used to hold the enrollment details of a zone.

```
typedef struct
{
    zuint8              u8ZoneID;
    zbmap16             u16ZoneType;
    zieeeaddress        u64IeeeAddress;
} tsCLD_IASACE_ZoneTable;
```

where:

- `u8ZoneID` is the identifier of the zone
- `u16ZoneType` is a value indicating the type of zone (for the possible values, refer to the description of the attribute `e16ZoneType` of the IAS Zone cluster in Section 37.2)
- `u64IeeeAddress` is the IEEE/MAC address of the device which hosts the zone

38.7.3 Zone Parameters

The following structure is used to store the ‘zone parameters’ on the IAS ACE cluster server.

```
typedef struct
{
    zbmap8                          u8ZoneConfigFlag;
    zbmap8                          u8ZoneStatusFlag;
    zbmap16                         eZoneStatus;
    zenum8                          eAudibleNotification;
    tsZCL_CharacterString           sZoneLabel;
    uint8
          au8ZoneLabel[CLD_IASACE_MAX_LENGTH_ZONE_LABEL];
    tsZCL_CharacterString           sArmDisarmCode;
    uint8
          au8ArmDisarmCode[CLD_IASACE_MAX_LENGTH_ARM_DISARM_CODE];
}tsCLD_IASACE_ZoneParameter;
```

where:

- `u8ZoneConfigFlag` is is a bitmap used to configure the temporal role of a zone (as Day, Night or Day/Night) and whether the zone is allowed to be bypassed. Macros are provided as follows:

Bit	Macro
0	CLD_IASACE_ZONE_CONFIG_FLAG_BYPASS *
1	CLD_IASACE_ZONE_CONFIG_FLAG_DAY_HOME
2	CLD_IASACE_ZONE_CONFIG_FLAG_NIGHT_SLEEP
3	CLD_IASACE_ZONE_CONFIG_FLAG_NOT_BYPASSED **
4-7	Reserved

* Determines whether the zone is allowed to be bypassed: 1 - allowed, 0 - not allowed

** Used to configure a status of ZONE_NOT_BYPASSED in responses to Bypass commands

`u8ZoneStatusFlag` is a bitmap used to indicate the current status of a zone as armed or bypassed. Macros are provided as follows:

Bit	Macro
0	CLD_IASACE_ZONE_STATUS_FLAG_BYPASS
1	CLD_IASACE_ZONE_STATUS_FLAG_ARM
2-7	Reserved

- `eZoneStatus` is the zone status as the value of the `b16ZoneStatus` attribute of the IAS Zone cluster (see Section 37.2)
- `eAudibleNotification` is a value specifying whether an audible notification (e.g. a chime) is required to signal a zone status change (enumerations are available in `teCLD_IASACE_AudibleNotification` - see Section 38.8.4):

Value	Status
0x00	Audible notification muted
0x01	Audible notification sounded
0x02 - 0xFF	Reserved

- `sZoneLabel` is the name/label for the zone represented as a character string
- `au8ZoneLabel[]` is the name/label for the zone represented as an array of ASCII values
- `sArmDisarmCode` is the arm/disarm code for the zone represented as a character string
- `au8ArmDisarmCode[]` is the arm/disarm code for the zone represented as an array of ASCII values

38.7.4 Panel Parameters

The following structure is used to store the ‘panel parameters’ on the IAS ACE cluster server.

```
typedef struct
{
    zenum8    ePanelStatus;
    zuint8    u8SecondsRemaining;
    zenum8    eAudibleNotification;
    zenum8    eAlarmStatus;
}tsCLD_IASACE_PanelParameter;
```

where:

- `ePanelStatus` is a value indicating the status to be displayed on the panel, as follows (enumerations are available in `teCLD_IASACE_PanelStatus` - see Section 38.8.2):

Value	Status
0x00	Disarmed (all zones) and ready to be armed
0x01	Armed stay
0x02	Armed night
0x03	Armed away
0x04	Exit delay
0x05	Entry delay

Value	Status
0x06	Not ready to be armed
0x07	In alarm
0x08	Arming stay
0x09	Arming night
0x0A	Arming away
0x0B - 0xFF	Reserved

- `u8SecondsRemaining` represents the time, in seconds, that the server will remain in the displayed state when the latter is ‘Exit delay’ or ‘Entry delay’ (for other states, this field should be set to 0x00).
- `eAudibleNotification` is a value specifying whether an audible notification (e.g. a chime) is required to signal a zone status change (enumerations are available in `teCLD_IASACE_AudibleNotification` - see Section 38.8.4):

Value	Status
0x00	Audible notification muted
0x01	Audible notification sounded
0x02 - 0xFF	Reserved

- `eAlarmStatus` is a value indicating the alarm status/type when the panel’s state is ‘In Alarm’, as follows (enumerations are available in `teCLD_IASACE_AlarmStatus` - see Section 38.8.3):

Value	Status
0x00	No alarm
0x01	Burglar
0x02	Fire
0x03	Emergency
0x04	Police panic
0x05	Fire panic
0x06	Emergency panic
0x07 - 0xFF	Reserved

38.7.5 Custom Command Payloads

The following structures contain the payloads for the IAS ACE cluster custom commands.

‘Arm’ Command Payload

The following structure contains the payload of a Arm command.

```
typedef struct
{
    zenum8                      eArmMode;
    tsZCL_CharacterString       sArmDisarmCode;
    zuint8                      u8ZoneID;
} tsCLD_IASACE_ArmPayload;
```

where:

- `eArmMode` is a value indicating the state of armament in which to put the zone (enumerations are available in `teCLD_IASACE_ArmMode` - see Section 38.8.1):

Value	Status
0x00	Disarm
0x01	Arm day/home zones only
0x02	Arm night/sleep zones only
0x03	Arm all zones
0x04 - 0xFF	Reserved

- `sArmDisarmCode` is an 8-character string containing the arm/disarm code (if a code is not required, set to "00000000")
- `u8ZoneID` is the identifier of the zone to arm/disarm

'Bypass' Command Payload

The following structure contains the payload of a Bypass command.

```
typedef struct
{
    zuint8                    u8NumOfZones;
    zuint8                   *pu8ZoneID;
    tsZCL_CharacterString     sArmDisarmCode;
} tsCLD_IASACE_BypassPayload;
```

where:

- `u8NumOfZones` is the number of zones to be 'bypassed' (taken out of the system)
- `pu8ZoneID` is a pointer to a list of identifiers specifying the zones to be bypassed (the number of zones in the list is specified in `u8NumOfZones`)
- `sArmDisarmCode` is an 8-character string containing the arm/disarm code (if a code is not required, set to "00000000")

'Get Zone Information' Command Payload

The following structure contains the payload of a Get Zone Information command.

```
typedef struct
{
    zuint8          u8ZoneID;
} tsCLD_IASACE_GetZoneInfoPayload;
```

where `u8ZoneID` is the identifier of the zone on which information is required.

'Set Bypassed Zone List' Command Payload

The following structure contains the payload of a Set Bypassed Zone List command.

```
typedef struct
{
    zuint8          u8NumofZones;
```

```
    zuint8          *pu8ZoneID;
} tsCLD_IASACE_SetBypassedZoneListPayload;
```

where:

- `u8NumofZones` is the number of zones in the new bypassed zone list
- `pu8ZoneID` is a pointer to the new bypassed zone list (the number of zones in the list is specified in `u8NumOfZones`)

‘Get Zone Status’ Command Payload

The following structure contains the payload of a Get Zone Status command.

```
typedef struct
{
    zuint8          u8StartingZoneID;
    zuint8          u8MaxNumOfZoneID;
    zbool           bZoneStatusMaskFlag;
    zbmap16         u16ZoneStatusMask;
} tsCLD_IASACE_GetZoneStatusPayload;
```

where:

- `u8StartingZoneID` is the identifier of the first zone for which status information is required
- `u8MaxNumOfZoneID` is the maximum number of zones for which status information should be returned
- `bZoneStatusMaskFlag` is a Boolean indicating whether status information should be returned for all zones or only for those zones with particular status values (specified through `u16ZoneStatusMask` below):
  - TRUE - only zones with specific status values
  - FALSE - all zones
- `u16ZoneStatusMask` is a 16-bit bitmap indicating the zone status values of interest (used when `bZoneStatusMaskFlag` is set to TRUE) - the response to the request will contain information only for those zones with a status value indicated in this bitmap:

Bit	Description
0	Alarm1: 1 - Opened or alarmed 0 - Closed or not alarned
1	Alarm2: 1 - Opened or alarmed 0 - Closed or not alarned
2	Tamper: 1 - Tampered with 0 - Not tampered with
3	Battery: 1 - Low 0 - OK
4	Supervision reports: 1 - Reports 0 - No reports
5	Restore reports: 1 - Reports

Bit	Description
	0 - No reports
6	Trouble: 1 - Trouble/failure 0 - OK
7	AC (mains): 1 - Fault 0 - OK
8	Test mode: 1 - Sensor in test mode 0 - Sensor in operational mode
9	Battery defect: 1 - Defective battery detected 0 - Battery OK
10-15	Reserved

‘Panel Status Changed or Get Panel Status Response’ Payload

The following structure contains the payload of a Panel Status Changed command or Get Panel Status Response.

```
typedef struct
{
    zenum8        ePanelStatus;
    zuint8        u8SecondsRemaining;
    zenum8        eAudibleNotification;
    zenum8        eAlarmStatus;
} tsCLD_IASACE_PanelStatusChangedOrGetPanelStatusRespPayload;
```

where:

- `ePanelStatus` is a value indicating the status to be displayed on the panel, as follows (enumerations are available in `teCLD_IASACE_PanelStatus` - see Section 38.8.2):

Value	Status
0x00	Disarmed (all zones) and ready to be armed
0x01	Armed stay
0x02	Armed night
0x03	Armed away
0x04	Exit delay
0x05	Entry delay
0x06	Not ready to be armed
0x07	In alarm
0x08	Arming stay
0x09	Arming night
0x0A	Arming away

Value	Status
0x0B - 0xFF	Reserved

- `u8SecondsRemaining` represents the time, in seconds, that the server will remain in the displayed state when the latter is ‘Exit delay’ or ‘Entry delay’ (for other states, this field should be set to 0x00).
- `eAudibleNotification` is a value specifying whether an audible notification (e.g. a chime) is required to signal a zone status change (enumerations are available in `teCLD_IASACE_AudibleNotification` - see Section 38.8.4):

Value	Status
0x00	Audible notification muted
0x01	Audible notification sounded
0x02 - 0xFF	Reserved

- `eAlarmStatus` is a value indicating the alarm status/type when the panel’s state is ‘In Alarm’, as follows (enumerations are available in `teCLD_IASACE_AlarmStatus` - see Section 38.8.3):

Value	Status
0x00	No alarm
0x01	Burglar
0x02	Fire
0x03	Emergency
0x04	Police panic
0x05	Fire panic
0x06	Emergency panic
0x07 - 0xFF	Reserved

38.7.6 Event Data Structures

The following structures hold the data contained in certain IAS ACE cluster events.

E_CLD_IASACE_CMD_ARM_RESP Data

```
typedef struct
{
    zenum8          eArmNotification;
} tsCLD_IASACE_ArmRespPayload;
```

where `eArmNotification` is an enumeration indicating the outcome of the Arm command, one of:

E_CLD_IASACE_ARM_NOTIF_ALL_ZONES_DISARMED
E_CLD_IASACE_ARM_NOTIF_ONLY_DAY_HOME_ZONES_ARMED
E_CLD_IASACE_ARM_NOTIF_ONLY_NIGHT_SLEEP_ZONES_ARMED
E_CLD_IASACE_ARM_NOTIF_ALL_ZONES_ARMED
E_CLD_IASACE_ARM_NOTIF_INVALID_ARM_DISARM_CODE
E_CLD_IASACE_ARM_NOTIF_NOT_READY_TO_ARM
E_CLD_IASACE_ARM_NOTIF_ALREADY_DISARMED

E_CLD_IASACE_CMD_GET_ZONE_ID_MAP_RESP Data

```
typedef struct
{
    zbmap16  au16ZoneIDMap[CLD_IASACE_MAX_BYTES_FOR_NUM_OF_ZONES];
} tsCLD_IASACE_GetZoneIDMapRespPayload;
```

where `au16ZoneIDMap[]` is an array, each element being a 16-bit bitmap indicating whether each of a set of zone identifiers is allocated - a Zone ID is represented by a single bit which is set to '1' if the identifier value has been allocated and '0' otherwise.

Array Element	Bit	Zone ID
au16ZoneIDMap[0]	0 1 : 15	0x00 0x01 : 0x0F
au16ZoneIDMap[1]	0 1 : 15	0x10 0x11 : 0x1F
:	:	:
au16ZoneIDMap[N]	0 1 : n : 15	16N 16N + 0x1 : 16N + 0xn : 16N + 0xF

E_CLD_IASACE_CMD_GET_ZONE_INFO_RESP Data

```
typedef struct
{
    zuint8                  u8ZoneID;
    zbmap16                 u16ZoneType;
    zieeeaddress            u64IeeeAddress;
    tsZCL_CharacterString   sZoneLabel;
} tsCLD_IASACE_GetZoneInfoRespPayload;
```

where:

- `u8ZoneID` is the identifier of the zone
- `u16ZoneType` is a value indicating the type of zone (for the possible values, refer to the description of the attribute `e16ZoneType` of the IAS Zone cluster in Section 37.2)
- `u64IeeeAddress` is the IEEE/MAC address of the device which hosts the zone
- `sZoneLabel` is a character string representing a name/label for the zone

E_CLD_IASACE_CMD_ZONE_STATUS_CHANGED Data

```
typedef struct
{
    zuint8                  u8ZoneID;
```

```
    zenum16                 eZoneStatus;
    zenum8                  eAudibleNotification;
    tsZCL_CharacterString   sZoneLabel;
} tsCLD_IASACE_ZoneStatusChangedPayload;
```

where:

- `u8ZoneID` is the identifier of the zone
- `u16ZoneType` is a value indicating the type of zone (for the possible values, refer to the description of the attribute `e16ZoneType` of the IAS Zone cluster in Section 37.2)
- `eAudibleNotification` is a value specifying whether an audible notification (e.g. a chime) to signal the change is required (enumerations are available in `teCLD_IASACE_AudibleNotification` - see Section 38.8.4):

Value	Status
0x00	Audible notification to be muted
0x01	Audible notification to be sounded
0x02 - 0xFF	Reserved

- `sZoneLabel` is a character string representing a name/label for the zone

E_CLD_IASACE_CMD_PANEL_STATUS_CHANGED Data

```
tsCLD_IASACE_PanelStatusChangedOrGetPanelStatusRespPayload
```

For details of this structure, see Section 38.7.5.

E_CLD_IASACE_CMD_GET_PANEL_STATUS_RESP Data

```
tsCLD_IASACE_PanelStatusChangedOrGetPanelStatusRespPayload
```

For details of this structure, see Section 38.7.5.

E_CLD_IASACE_CMD_BYPASS_RESP Data

```
typedef struct
{
    zuint8          u8NumofZones;
    zuint8         *pu8BypassResult;
} tsCLD_IASACE_BypassRespPayload;
```

where:

- `u8NumOfZones` is the number of zones ‘bypassed’ (taken out of the system)
- `pu8BypassResult` is a pointer to a list of identifiers specifying the zones bypassed (the number of zones in the list is specified in `u8NumOfZones`)

E_CLD_IASACE_CMD_GET_ZONE_STATUS_RESP Data

```
typedef struct
{
    zbool           bZoneStatusComplete;
```

```
    zuint8          u8NumofZones;
    zuint8         *pu8ZoneStatus;
} tsCLD_IASACE_GetZoneStatusRespPayload;
```

where:

- `bZoneStatusComplete` is a Boolean indicating whether the current response completes the set of zones for which status information can be returned (if not, the client should send another Get Zone Status command to the server):
  - TRUE - no more zone status information to be returned
  - FALSE - status information for more zones available to be queried
- `u8NumofZones` is the number of zones for which status information was returned in this response
- `pu8ZoneStatus` is a pointer to a list of status values for the reported zones (the number of values in the list is indicated by `u8NumofZones` above) - each is a 24-bit value containing the following information:

Bits	Description
0-7	Zone ID
8-23	Value of `b16ZoneStatus` attribute of the IAS Zone cluster for the zone

38.8 Enumerations

38.8.1 teCLD_IASACE_ArmMode

The following structure contains the enumerations used to indicate a mode of armament:

```
typedef enum
{
    E_CLD_IASACE_ARM_MODE_DISARM = 0x00,
    E_CLD_IASACE_ARM_MODE_ARM_DAY_HOME_ZONES_ONLY,
    E_CLD_IASACE_ARM_MODE_ARM_NIGHT_SLEEP_ZONES_ONLY,
    E_CLD_IASACE_ARM_MODE_ARM_ALL_ZONES,
} teCLD_IASACE_ArmMode;
```

38.8.2 teCLD_IASACE_PanelStatus

The following structure contains the enumerations used to indicate the status of the panel:

```
typedef enum
{
    E_CLD_IASACE_PANEL_STATUS_PANEL_DISARMED = 0x00,
    E_CLD_IASACE_PANEL_STATUS_PANEL_ARMED_DAY,
    E_CLD_IASACE_PANEL_STATUS_PANEL_ARMED_NIGHT,
    E_CLD_IASACE_PANEL_STATUS_PANEL_ARMED_AWAY,
    E_CLD_IASACE_PANEL_STATUS_PANEL_EXIT_DELAY,
    E_CLD_IASACE_PANEL_STATUS_PANEL_ENTRY_DELAY,
    E_CLD_IASACE_PANEL_STATUS_PANEL_NOT_READY_TO_ARM,
    E_CLD_IASACE_PANEL_STATUS_PANEL_IN_ALARM,
    E_CLD_IASACE_PANEL_STATUS_PANEL_ARMING_STAY,
    E_CLD_IASACE_PANEL_STATUS_PANEL_ARMING_NIGHT,
    E_CLD_IASACE_PANEL_STATUS_PANEL_ARMING_AWAY
} teCLD_IASACE_PanelStatus;
```

38.8.3 teCLD_IASACE_AlarmStatus

The following structure contains the enumerations used to indicate the status/meaning of the alarm:

```
typedef enum
{
    E_CLD_IASACE_ALARM_STATUS_NO_ALARM = 0x00,
    E_CLD_IASACE_ALARM_STATUS_BURGLAR,
    E_CLD_IASACE_ALARM_STATUS_FIRE,
    E_CLD_IASACE_ALARM_STATUS_EMERGENCY,
    E_CLD_IASACE_ALARM_STATUS_POLICE_PANIC,
    E_CLD_IASACE_ALARM_STATUS_FIRE_PANIC,
    E_CLD_IASACE_ALARM_STATUS_EMERGENCY_PANIC
} teCLD_IASACE_AlarmStatus;
```

38.8.4 teCLD_IASACE_AudibleNotification

The following structure contains the enumerations used to indicate the configuration of the audible indication:

```
typedef enum
{
    E_CLD_IASACE_AUDIBLE_NOTIF_MUTE = 0x00,
    E_CLD_IASACE_AUDIBLE_NOTIF_DEFAULT_SOUND
} teCLD_IASACE_AudibleNotification;
```

38.9 Compile-time options

To enable the IAS ACE cluster in the code to be built, it is necessary to add the following to the **zcl_options.h** file:

```
#define CLD_IASACE
```

In addition, to include the software for a cluster client or server or both, it is necessary to add one of the following to the same file:

```
#define IASACE_SERVER
#define IASACE_CLIENT
```

The IAS ACE cluster contains macros that may be specified at compile-time by adding one or more of the following lines to the **zcl_options.h** file.

Maximum Size of Zone Table

The maximum number of entries in a Zone table on the cluster server can be defined using the following line:

```
#define CLD_IASACE_ZONE_TABLE_SIZE n
```

where n is the desired maximum (e.g. 8).

Maximum Length of Arm/Disarm Code

The maximum length of string allowed for the arm/disarm code can be defined using the following line:

```
#define CLD_IASACE_MAX_LENGTH_ARM_DISARM_CODE n
```

where n is the desired maximum.

Maximum Length of Zone Label

The maximum length of string allowed for a zone name/label can be defined using the following line:

```
#define CLD_IASACE_MAX_LENGTH_ZONE_LABEL n
```

where n is the desired maximum.

Disable APS Acknowledgements for Bound Transmissions

APS acknowledgements for bound transmissions from this cluster can be disabled using the following line:

```
#define CLD_IASACE_BOUND_TX_WITH_APS_ACK_DISABLED
```

Cluster Revision

Add this line to define the value (n) of the Cluster Revision attribute:

```
#define CLD_IASACE_CLUSTER_REVISION <n>
```

The default value is 1, which corresponds to the revision of the cluster in the ZCL r6 specification (see Section 2.4).

39 IAS Warning Device Cluster

This chapter describes the IAS Warning Device (WD) cluster which provides an interface to a Warning Device in an IAS (Intruder Alarm System).

The IAS WD cluster has a Cluster ID of 0x0502.

39.1 Overview

The IAS WD cluster provides an interface to an IAS Warning Device, allowing warning indications triggered by alarm conditions to be sent to it. The server side of the cluster is implemented on the IAS Warning Device and the client side is implemented on the triggering device. The IAS Warning Device is detailed in the *ZigBee Devices User Guide (JNUG3131)*.

To use the functionality of this cluster, you must include the file **IASWD.h** in your application and enable the cluster by defining CLD_IASWD in the **zcl_options.h** file.

The inclusion of the client or server software must be pre-defined in the compile-time options of the application. In addition, if the cluster resides on a custom endpoint, then the role of client or server must also be specified when creating the cluster instance.

The compile-time options for the IAS WD cluster are fully detailed in Section 39.7.

39.2 IAS WD Structure and Attribute

The structure definition for the IAS WD cluster is:

```
typedef struct
{
#ifdef IASWD_SERVER
  zuint16   u16MaxDuration;
#endif
  zuint16   u16ClusterRevision;
} tsCLD_IASWD;
```

where:

- `u16MaxDuration` is the maximum duration, in seconds, for which the alarm can be continuously active (for example, a siren sounded). The range of possible values is 0 to 65534 seconds and the default value is 240 seconds.
- `u16ClusterRevision` is a mandatory attribute that specifies the revision of the cluster specification on which this cluster instance is based. The cluster specification in the ZCL r6 corresponds to a cluster revision of 1. The value is incremented by one for each subsequent revision of the cluster specification. This attribute is also described in Section 2.4.

39.3 Issuing Warnings

The IAS WD cluster allows a device which detects warning conditions (example, fire) to trigger a warning on an IAS Warning Device which, in turn, initiates a physical alarm such as a siren and/or strobe. The IAS Warning Device hosts the cluster server and the triggering device hosts the cluster client.

Two types of warning can be initiated:

- **Warning mode:** This mode indicates a genuine emergency, such as a fire or an intruder. On detection of the emergency condition, the application on the triggering device must call the **eCLD_IASWDStartWarningReqSend()** function, which sends a Start Warning command to the Warning Device. The payload of this command contains the time-duration for which the Warning

Device must remain in warning mode. The specified duration must not exceed the maximum duration defined in the `u16MaxDuration` attribute on the Warning Device (see Section 39.2). The payload also contains details of the warning and the strobe requirements, if any. On receiving this command, an E_CLD_IASWD_CMD_WD_START_WARNING event is generated on the Warning Device (see Section 39.4) for the attention of the application.

- **Squawk mode:** This mode indicates a change of state of the IAS system - that is, armed or disarmed. Thus, this is typically a short audible beep or 'squawk' that is emitted when the system is armed or disarmed. To initiate a squawk, the application on the triggering device must call the function **eCLD_IASWDSquawkReqSend()**, which sends a Squawk command to the Warning Device. The payload also contains details of the squawk and the strobe requirements, if any. On receiving this command, an E_CLD_IASWD_CMD_WD_SQUAWK event is generated on the Warning Device (see Section 39.4) for the attention of the application.

The payloads of the commands are detailed in Section 39.6.2.

Note: *In order to maintain timing information on the cluster server, the application on the Warning Device must periodically call the* ***eCLD_IASWDUpdate()*** *function every 100 ms. These calls can be prompted using a software timer.*

Note: *The `u16MaxDuration` attribute on the Warning Device can be updated by the application on this device by calling the function* ***eCLD_IASWDUpdateMaxDuration()****.*

39.4 IAS WD Events

The IAS WD cluster has its own events that are handled through the callback mechanism outlined in Chapter 3. If a device uses the IAS WD cluster then IAS WD 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 **eHA_RegisterWarningDeviceEndPoint()** for a Warning Device). The relevant callback function is then invoked when an IAS WD event occurs.

For an IAS WD 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 `tsCLD_IASWDCallBackMessage` structure:

```
typedef struct
{
  uint8   u8CommandId;
  union
  {
  tsCLD_IASWD_StartWarningReqPayload  *psWDStartWarningReqPayload;
  tsCLD_IASWD_SquawkReqPayload   *psWDSquawkReqPayload;
  tsCLD_IASWD_StrobeUpdate  *psStrobeUpdate;  /* Internal */
  tsCLD_IASWD_WarningUpdate   *psWarningUpdate; /* Internal */
  } uMessage;
} tsCLD_IASWDCallBackMessage;
```

When an IAS WD event occurs, one of several command types could have been received. The relevant command type is specified through the `u8CommandId` field of the `tsSM_CallBackMessage` structure. The possible command/event types are detailed in the table below (not that `psStrobeUpdate` and `psWarningUpdate` are for internal use only).

Table 70. IAS WD Command Types

u8CommandId Enumeration	Description
E_CLD_IASWD_CMD_WD_START_WARNING	A Start Warning command has been received by the cluster server - this command requests that the alarm is activated for a specified time. The

Table 70. IAS WD Command Types...*continued*

u8CommandId Enumeration	Description
	command payload is contained in the event in the `tsCLD_IASWD_StartWarningReqPayload` structure, described in Section 39.6.2.
E_CLD_IASWD_CMD_WD_SQUAWK	A Squawk command has been received by the cluster server - this command requests that the alarm is briefly activated to emit a 'squawk' to indicate a status change, such as system disarmed. The command payload is contained in the event in the `tsCLD_IASWD_SquawkReqPayload` structure, described in Section 39.6.2.

39.5 Functions

The following IAS WD cluster functions are provided in the NXP implementation of the ZCL:

1. eCLD_IASWDCreateIASWD
2. eCLD_IASWDUpdate
3. eCLD_IASWDUpdateMaxDuration
4. eCLD_IASWDStartWarningReqSend
5. eCLD_IASWDSquawkReqSend

39.5.1 eCLD_IASWDCreateIASWD

```
teZCL_Status eCLD_IASWDCreateIASWD(
  tsZCL_ClusterInstance *psClusterInstance,
  bool_t bIsServer,
  tsZCL_ClusterDefinition *psClusterDefinition,
  void *pvEndPointSharedStructPtr,
  uint8 *pu8AttributeControlBits,
  tsCLD_IASWDCustomDataStructure
          *psCustomDataStructure);
```

Description

This function creates an instance of the IAS WD 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 creates an IAS WD cluster instance on the endpoint, but instances of other clusters may also be created on the same endpoint by calling their corresponding creation functions.

Note: *This function must not be called for an endpoint on which a standard ZigBee device is used. In this case, the device and its supported clusters must be registered on the endpoint using the relevant device registration function.*

When used, this function must be called after the stack has been started and after the ZCL has been initialized.

Parameters

- *psClusterInstance*: Pointer to structure containing information about the cluster instance to be created (see Section 6.1.16). This structure is updated by the function by initializing individual structure fields.
- *bIsServer*: Type of cluster instance (server or client) to be created:
- TRUE - server
- FALSE - client
- *psClusterDefinition*: Pointer to structure indicating the type of cluster to be created (see Section 6.1.2). In this case, this structure must contain the details of the IAS WD cluster. This parameter can refer to a pre-filled structure called `sCLD_IASWD` which is provided in the **IASWarningDevice.h** file.
- *pvEndPointSharedStructPtr*: Pointer to the shared structure used for attribute storage. This parameter should be the address of the structure of type `tsCLD_IASWD` which defines the attributes of IAS WD cluster. The function initializes the attributes with default values.
- *pu8AttributeControlBits*: Pointer to an array of **uint8** values, with one element for each attribute in the cluster.
- *psCustomDataStructure*: Pointer to a structure containing the storage for internal functions of the cluster (see Section 39.6.1)

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_INVALID_VALUE

39.5.2 eCLD_IASWDUpdate

```
teZCL_Status eCLD_IASWDUpdate(
  uint8 u8SourceEndPoint);
```

Description

This function can be used on an IAS WD cluster server to update the timing requirements of the Warning Device. The function should be called by the application at a rate of once every 100 ms.

Parameters

- *u8SourceEndPointId*: Number of the endpoint on which the IAS WD cluster resides

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL

39.5.3 eCLD_IASWDUpdateMaxDuration

```
teZCL_Status eCLD_IASWDUpdateMaxDuration(
  uint8 u8SourceEndPointId,
  uint16 u16MaxDuration);
```

Description

This function can be used on an IAS WD cluster server to set the value of the `u16MaxDuration` attribute, which represents the maximum duration, in seconds, for which the alarm can be continuously active.

The set value is the maximum duration, in seconds, for which the alarm can be active following a received Start Warning request.

Parameters

- *u8SourceEndPointId*: Number of the endpoint on which the IAS WD cluster resides
- *u16MaxDuration*: Value to which attribute will be set, in the range 0 to 65534

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL

39.5.4 eCLD_IASWDStartWarningReqSend

```
teZCL_Status eCLD_IASWDStartWarningReqSend(
  uint8 u8SourceEndPointId,
  uint8 u8DestinationEndPointId,
  tsZCL_Address *psDestinationAddress,
  uint8 *pu8TransactionSequenceNumber,
  tsCLD_IASWD_StartWarningReqPayload *psPayload);
```

Description

This function can be used on IAS WD cluster client to send a Start Warning command to the IAS WD server on a Warning Device.

The receiving IAS WD server activates the alarm on the Warning Device for a specified duration.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is 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 to send the request. This parameter is used both to send the command and to identify the instance of the shared structure holding the required attribute values
- *u8DestinationEndPointId*: Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP
- *psDestinationAddress*: Pointer to a structure holding the address of the node to which the request is sent
- *pu8TransactionSequenceNumber*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the request
- *psPayload*: Pointer to a structure containing the payload for the command (see Section 39.6.2)

Returns

- E_ZCL_SUCCESS

- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL
- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

39.5.5 eCLD_IASWDSquawkReqSend

```
teZCL_Status eCLD_IASWDSquawkReqSend(
 uint8 u8SourceEndPointId,
   uint8 u8DestinationEndPointId,
  tsZCL_Address *psDestinationAddress,
   uint8 *pu8TransactionSequenceNumber,
   tsCLD_IASWD_SquawkReqPayload *psPayload);
```

Description

This function can be used on IAS WD cluster client to send a Squawk command to the IAS WD server on a Warning Device.

The receiving IAS WD server will briefly activate the alarm on the Warning Device to emit a ‘squawk’ - depending on the device, this could be a visible and/or audible emission. The parameters of the squawk are specified in the command payload.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is 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 to send the request. This parameter is used both to send the command and to identify the instance of the shared structure holding the required attribute values
- *u8DestinationEndPointId*: Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP
- *psDestinationAddress*: Pointer to a structure holding the address of the node to which the request is sent
- *pu8TransactionSequenceNumber*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the request
- *psPayload*: Pointer to a structure containing the payload for the command (see Section 39.6.2)

Returns

- E_ZCL_SUCCESS
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_EP_UNKNOWN
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_ZBUFFER_FAIL

- E_ZCL_ERR_ZTRANSMIT_FAIL

If an error is returned by the ZigBee PRO stack function which is invoked by this function to transmit the data, this error may be obtained by calling **eZCL_GetLastZpsError()**.

39.6 Structures

39.6.1 Custom Data Structure

The IAS WD cluster requires extra storage space to be allocated to be used by internal functions. The structure definition for this storage is shown below:

```
typedef struct
{
 tsCLD_IASWD_SquawkReqPayload sSquawk;
 tsCLD_IASWD_StartWarningReqPayload sWarning;
 uint32   u32WarningDurationRemainingIn100MS;
 tsZCL_ReceiveEventAddress  sReceiveEventAddress;
 tsZCL_CallBackEvent  sCustomCallBackEvent;
 tsCLD_IASWDCallBackMessage sCallBackMessage;
} tsCLD_IASWD_CustomDataStructure;
```

The fields are for internal use and no knowledge of them is required.

39.6.2 Custom Command Payloads

The following structures contain the payloads for the IAS WD cluster custom commands.

‘Start Warning’ Payload

The following structure contains the payload of a Start Warning command.

```
typedef struct
{
 uint8   u8WarningModeStrobeAndSirenLevel;
 uint16  u16WarningDuration;
 uint8   uStrobeDutyCycle;
 enum8       eStrobeLevel;
}tsCLD_IASWD_StartWarningReqPayload;
```

where:

- `u8WarningModeStrobeAndSirenLevel` is an 8-bit bitmap containing the requirements for the warning alarm, as follows:

Bits	Description
0-3	Warning Mode - indicates the meaning of the requested warning: 0 - Stop (no warning) 1 - Burglar 2 - Fire 3 - Emergency 4 - Police panic 5 - Fire panic 6 - Emergency (medical) panic All other values are reserved

Bits	Description
4-5	Strobe* - indicates whether a visual strobe indication of the warning is required: 0 - No strobe 1 - Use strobe Other values are reserved
6-7	Siren Level - indicates the requested level of an audible siren (if enabled): 0 - Low level 1 - Medium level 2 - High level 3 - Very high level

* If ‘Strobe’ is 1 and ‘Warning Mode’ is 0, only the strobe is activated

- `u16WarningDuration` is the requested time-duration of the warning, in seconds, which must be less than or equal to the value of the `u16MaxDuration` attribute
- `uStrobeDutyCycle` is the duty-cycle of the strobe pulse, expressed as a percentage in 10% steps (example, 0x1E represents 30%) - invalid values are rounded to the nearest multiple of 10%
- `eStrobeLevel` is the level of the strobe (pulse)

‘Squawk’ Payload

The following structure contains the payload of a Squawk command.

```
typedef struct
{
  uint8  u8SquawkModeStrobeAndLevel;
}tsCLD_IASWD_SquawkReqPayload;
```

Where `u8SquawkModeStrobeAndLevel` is an 8-bit bitmap containing the requirements for the ‘squawk’, as follows.

Bits	Description
0-3	Squawk Mode - indicates the meaning of the required ‘squawk’: 0 - System is armed 1 - System is disarmed All other values are reserved
4	Strobe - indicates whether a visual strobe indication of the ‘squawk’ is required: 0 - No strobe 1 - Use strobe
5	Reserved
6-7	Squawk Level - indicates the requested level of the audible squawk sound: 0 - Low level 1 - Medium level 2 - High level 3 - Very high level

39.6.3 Event Data Structures

The following structures hold the data contained in certain IAS WD cluster events.

E_CLD_IASWD_CLUSTER_UPDATE_STROBE Data

```
typedef struct
{
 bool_t   bStrobe;
 uint8    u8StrobeDutyCycle;
 zenum8   eStrobeLevel;
}tsCLD_IASWD_StrobeUpdate;
```

where:

```
bStrobe is the current (new) status of the strobe:
```

- TRUE - Strobe ‘on’
- FALSE - Strobe ‘off’
  - `uStrobeDutyCycle` is the duty-cycle of the strobe pulse, expressed as a percentage in 10% steps (example, 0x1E represents 30%) - invalid values is rounded to the nearest multiple of 10%
  - `eStrobeLevel` is the level (brightness) of the strobe pulse:
- 0 - Low level
- 1 - Medium level
- 2 - High level
- 3 - Very high level

  All other values are reserved

E_CLD_IASWD_CLUSTER_UPDATE_WARNING Data

```
typedef struct
{
 uint8    u8WarningMode;
 uint16   u16WarningDurationRemaining;
 zenum8   eStrobeLevel;
}tsCLD_IASWD_WarningUpdate;
```

where:

- `u8WarningMode` is a value indicating the current warning mode:
  - 0 - No warning
  - 1 - Burglar
  - 2 - Fire
  - 3 - Emergency
  - 4 - Police panic
  - 5 - Fire panic
  - 6 - Emergency (medical) panic

    All other values are reserved
- `u16WarningDurationRemaining` is the time, in seconds, during which the device remains in warning mode
- `eStrobeLevel` is the level of the strobe (pulse)

39.7 Compile-time Options

To enable the IAS WD cluster in the code to be built, it is necessary to add the following to the **zcl_options.h** file:

```
#define CLD_IASWD
```

In addition, to include the software for a cluster client or server or both, it is necessary to add one of the following to the same file:

```
#define IASWD_SERVER
#define IASWD_CLIENT
```

The IAS WD cluster contains macros that may be specified at compile-time by adding one or more of the following lines to the **zcl_options.h** file.

Add this line to define the value (n) of the Cluster Revision attribute:

```
#define CLD_IASWD_CLUSTER_REVISION <n>
```

The default value is 1, which corresponds to the revision of the cluster in the ZCL r6 specification (see Section 2.4).

Part IX: Smart Energy Clusters

This part comprises three chapters:

- Chapter 40 details the **Price** cluster
- Chapter 41 details the **Demand-Response and Load Control** cluster
- Chapter 42 details the **Simple Metering** cluster

40 Price Cluster

This chapter outlines the Price cluster, which is used to hold and exchange price information.

The Price cluster has a Cluster ID of 0x0700.

CAUTION: ***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.*

40.1 Overview

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

Table 71. Price Cluster in ZigBee Devices

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

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. Further compile-time options for the Price cluster are detailed in Section 40.13.

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

Functions are provided for implementing the cluster commands. These functions are referenced in Section 40.4 and Section 40.5, and detailed in Section 40.9.

40.2 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:

40.2.1 '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.*

40.2.2 '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]
```

40.2.3 '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`

40.2.4 '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 42.10.6
  - `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 40.11.1 (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.

40.2.5 '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]
```

40.2.6 '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
```

40.2.7 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 Table74.

Note: *Price information for Time-Of-Use (TOU) mode is held in the `tsSE_PricePublishPriceCmdPayload` structure described in Section 40.11.1. Prices are matched to tiers using the strings defined in the Tier Label attributes.*

40.3 Attribute settings

The Price cluster structure (see Section 40.2) 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 depend on the number of tiers implemented (and Block mode attributes must be disabled).

Note:

- *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 42.2.*
- *The price information for Time-Of-Use (TOU) mode is stored in the structure `tsSE_PricePublishPriceCmdPayload` described in Section 40.11.1.*

40.4 Initializing 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 startup, the Price cluster software initializes 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

automatically adds this price information to its own price list (see Section 40.5.1). However, at ESP startup, 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 is, therefore, 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 40.5.3.

Note: *When initializing the price list at ESP startup, 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 initialization, 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, it is 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 40.5.

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 synchronized, i.e. a call to **bZCL_GetTimeHasBeenSynchronised()** returns TRUE

In addition to the function **eSE_PriceAddPriceEntry()**, the following functions 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 40.9.

40.5 Publishing price information

This section and its sub-sections describe the ways in which price information can be published (distributed) in a ZigBee network. As introduced in Section 40.1, 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 40.5.1
- Response to a Get Current Price command - refer to Section 40.5.2
- Response to a Get Scheduled Prices command - refer to Section 40.5.3

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 (for example, gas), unit of measure, currency, price, current time, start-time, and duration. On receipt of this command, if valid, the received price information

is automatically added to the price list on the device. If it is successfully added, an E_SE_PRICE_TABLE_ADD event is generated on the receiving device and this event is handled by the callback function registered for the relevant endpoint.

40.5.1 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 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 3.0 Stack User Guide (JNUG3130).*

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 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 40.5.2). 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 40.8). This command can be used by devices that do not implement a real-time clock.

40.5.2 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 function **eSE_PriceGetCurrentPriceSend()** 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.

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 40.5.1). 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.

40.5.3 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 function **eSE_PriceGetScheduledPricesSend()** 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.

40.6 Time-synchronization via Publish Price commands

As an alternative to using the Time cluster to time-synchronize a ZigBee device with the ESP (as described in Section 18.5.3), the local application can use the time embedded in a Publish Price command from the ESP (see Section 40.5), as described below.

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

It is the responsibility of the application on a ZigBee device to perform time-synchronization with the ESP. This involves updating the ZCL time on the local device.

The initialization 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 18.5.2 (this method also gets the time-zone and daylight saving information).

Subsequent re-synchronizations 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 a ZCL user task to be activated. The event is initially handled by this task as described in Chapter 3, 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 Chapter 3) which, provided that the event is of the type E_SE_PRICE_TIME_UPDATE, must update the ZCL time using **vZCL_SetUTCTime()**.

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-synchronization 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 18.5.3.

It is worth noting that an undefined ZCL time causes the following issues in the Price cluster:

- 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-synchronized with the ESP (as described in Section 18.5.2) 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.

40.7 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 40.2.

If required, conversion factor and/or calorific value must be enabled in the compile-time options, as described in Section 40.13.

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.

Initializing Conversion Factors and Calorific Values at Network Startup

Note the following issues at network startup:

- 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 initializing the conversion factor or calorific value at ESP startup, 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 initialization 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 (example, **eSE_RegisterEspMeterEndPoint()**).

40.8 Price events

The Price cluster has its own events that are handled through the callback mechanism described in Chapter 3. 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 40.12.2 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 40.5.1 and Section 40.5.2.

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 40.6.

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.

40.9 Functions

The following Price cluster functions are provided:

- eSE_PriceCreate
- eSE_PriceGetCurrentPriceSend
- eSE_PriceGetScheduledPricesSend
- eSE_PriceAddPriceEntry
- eSE_PriceAddPriceEntryToClient
- eSE_PriceGetPriceEntry
- eSE_PriceDoesPriceEntryExist
- eSE_PriceRemovePriceEntry
- eSE_PriceClearAllPriceEntries
- eSE_PriceAddConversionFactorEntry
- eSE_PriceGetConversionFactorSend
- eSE_PriceGetConversionFactorEntry
- eSE_PriceDoesConversionFactorEntryExist
- eSE_PriceRemoveConversionFactorEntry
- eSE_PriceClearAllConversionFactorEntries
- eSE_PriceAddCalorificValueEntry
- eSE_PriceGetCalorificValueSend
- eSE_PriceGetCalorificValueEntry
- eSE_PriceDoesCalorificValueEntryExist
- eSE_PriceRemoveCalorificValueEntry
- eSE_PriceClearAllCalorificValueEntries

40.9.1 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 D.

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 the ZigBee Devices User Guide*

Note: *(JNUG3131).*

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 ZCL has been initialized.

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 PRICE_NUM_OF_ATTRIBUTES.

The array declaration should be as follows:

```
uint8 au8AppPriceClusterAttributeControlBits[PRICE_NUM_OF_ATTRIBUTES];
```

The function initializes 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 Section 6.1.16). This structure is updated by the function by initialising individual structure fields.

- *psClusterDefinition* Pointer to structure indicating the type of cluster to be created (see Section 6.1.2). 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 initializes the attributes with default values.

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_INVALID_VALUE

40.9.2 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 is 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 is sent
- *u8DestinationEndPointId* Number of the remote endpoint to which the request is sent
- *psDestinationAddress* Pointer to a structure containing the address of the remote node to which the request is 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 (for example, 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

40.9.3 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 is 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 is sent
- *u8DestinationEndPointId* Number of the remote endpoint to which the request is sent
- *psDestinationAddress* Pointer to a structure containing the address of the remote node to which the request is 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

40.9.4 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 is 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 is sent
- *u8DestinationEndPointId* Number of the remote endpoint to which the request is sent
- *psDestinationAddress* Pointer to a structure containing the address of the remote node to which the Publish Price command is 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 40.11.1). 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

40.9.5 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 is 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 40.11.1). 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

40.9.6 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 (for example, on ESP)
  - FALSE - client (for example, 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 40.11.1), 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

40.9.7 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

40.9.8 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

40.9.9 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

40.9.10 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 is 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 is sent
- *u8DestinationEndPointId* Number of the remote endpoint to which the request is sent
- *psDestinationAddress* Pointer to a structure containing the address of the remote node to which the request is 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

40.9.11 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 is 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 is sent
- *u8DestinationEndPointId* Number of the remote endpoint to which the request is sent
- *psDestinationAddress* Pointer to a structure containing the address of the remote node to which the request is 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

40.9.12 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 (example on ESP)
  - FALSE - client (example 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 40.11.2), 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

40.9.13 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

40.9.14 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

40.9.15 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

40.9.16 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 is 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 is sent
- *u8DestinationEndPointId*: Number of the remote endpoint to which the request is sent
- *psDestinationAddress*: Pointer to a structure containing the address of the remote node to which the request is 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

40.9.17 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 is 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 is sent
- *u8DestinationEndPointId*: Number of the remote endpoint to which the request is sent
- *psDestinationAddress*: Pointer to a structure containing the address of the remote node to which the request is 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

40.9.18 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 (example on ESP)
  - FALSE - client (example 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 40.11.3), 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

40.9.19 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

40.9.20 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

40.9.21 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

40.10 Return codes

In addition to some of the ZCL status enumerations, the following enumerations are returned by Price cluster functions (see Section 40.9) 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.

Table 72. Price Cluster Return Codes

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 *

Table 72. Price Cluster Return Codes*...continued*

Enumeration	Description
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)
E_SE_PRICE_BLOCK_PERIOD_TA-BLE_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)

* Value of `u32IssuerEventId` in `tsSE_PricePublishPriceCmdPayload` structure (see Section 40.11.1) is less for the price to be added than for the existing (overlapping) price.

40.11 Structures

40.11.1 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 42.10.3)
- `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

40.11.2 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

40.11.3 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 40.12.3.
- `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

40.12 Enumerations

40.12.1 '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;
```

40.12.2 '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 56 below.

Note: *For further details on Price events, refer to Section 40.8.*

Table 73. Price Event Types

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

Table 73. Price Event Types*...continued*

Event Type Enumeration	Description
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

40.12.3 '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 57 below.

Table 74. 'Calorific Value Unit' Enumerations

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

40.13 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

41 Demand-Response and Load Control Cluster

This chapter outlines the Demand-Response and Load Control (DRLC) cluster. 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.

41.1 Overview

The DRLC cluster is required in ZigBee devices as indicated in the table below.

Table 75. DRLC Cluster in ZigBee Devices

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

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, 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. Further compile-time options for the DRLC cluster are detailed in Section 41.12.

41.2 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 (for example, water heater and pool pump). Enumerations are provided for the device classes and are detailed in Section 41.10.1. 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.*

41.3 Initialization

Provided that the DRLC cluster is enabled in the compile-time options (see Section 41.12), the cluster is automatically initialized when the ZCL is initialized and the ZigBee device is registered in the application - that is, by calling **eZCL_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 initialization, 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 initialization 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 41.2) 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 3.0 Stack User Guide (JNUG3130)*.

3. **Synchronize time with ESP:** A client should synchronize ZCL time with the ESP using the Time cluster as soon as initialization is complete. It is not possible to process unsolicited LCEs with a ‘start-time of now’ until ZCL time has been synchronized.

Once the clients have been set up, the ESP/server may need to configure the enrolment groups and randomization attributes of the DRLC clients (see Section 41.2). 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

41.4 Load Control Events (LCEs)

The 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 41.4.1.

An LCE is sent from the utility company to the DRLC server (ESP) of a ZigBee network, from where it is passed to DRLC clients. The LCEs are held in lists on the server and clients, as described in Section 41.4.2.

LCE handling is described in Section 41.5.

41.4.1 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)
- Randomization 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 41.11.1.

41.4.2 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 canceled with a randomized 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 randomized start-time). At the end of an active LCE, it is automatically moved to the `Deallocated list`. However, an active LCE which is canceled 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 41.5.1, but is done automatically by the cluster on the clients. All other operations on LCE lists, apart from cancelation (see Section 41.5.3), 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

41.5 LCE Handling

LCEs are handled on the DRLC cluster server and clients as described in Section 41.5.1 and Section 41.5.2 respectively. Canceling LCEs is described in Section 41.5.3.

Note: *The DRLC callback events referred to in this section are further described in Section 41.7 and are handled by the callback function that is registered as part of the device endpoint registration.*

41.5.1 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 41.3).

Note:

1. ***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()****.*
2. ***Note 2:*** *On receiving an LCE, the client checks the device class and enrollment group specified within the LCE, and only accepts the LCE if these values match the corresponding DRLC cluster attributes held locally (see Section 41.2).*

The cluster server also automatically responds to Get Scheduled Events messages from cluster clients that need current and future LCEs (see Section 41.5.2.2).

41.5.2 LCE Handling on Clients

The sub-sections below describe the various LCE handling activities that take place on a DRLC cluster client.

41.5.2.1 LCE Activation and De-activation

On receiving a new LCE from the DRLC cluster server, a cluster client first checks the device class and enrollment group specified within the LCE. If they do not match those of the local device (see DRLC attributes in Section 41.2), 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 randomized (see below).

If a new LCE is successfully added to the Scheduled (or Active) list, the client sends 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 randomized 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 41.2).

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 randomized 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 41.2).

Note: *The above randomize attributes of the DRLC cluster also allow LCE start-time and end-time randomization to be disabled for all LCEs on the local device. If this is the case, randomization settings within the LCE itself are ignored.*

41.5.2.2 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 are 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 results 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 responds 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.

41.5.2.3 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 41.4.2). Details of the actioned LCE are sent in a `tsSE_DRLCReportEvent` structure (see Section 41.11.4). The nature of the action is indicated in this structure using an enumeration (see Section 41.10.8).

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.

41.5.2.4 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°C, but the user may choose to over-ride this with a cooling level of 20°C. The above data values and their formats are detailed in the LCE structure description in Section 41.11.1.

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.

41.5.3 Canceling LCEs

An LCE can be canceled, 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 cancelations:

- **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 indicates whether the cancellation with immediate effect or a random delay is 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 puts the LCE in the 'Cancelled' list until the delay has expired, when the LCE is 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 is then generated, this time indicating immediate cancellation. The application should now stop load control for the relevant device(s).

41.6 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.*
2. *Signature fields are included in the Report Event Status structure, detailed in Section 41.11.4.*

Message signing must be enabled at compile-time, as described in Section 41.12.

41.7 DRLC Events

The DRLC cluster has its own events that are handled through the callback mechanism described in Chapter 3. 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 is then 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.

41.7.1 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:

Table 76. Command Types

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 randomized end-time is specified in the `uMessage.sCancelLoadControlEvent` field of the `tsSE_DRLCCallBackMessage` structure
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 randomized 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

* If an LCE cancellation with a randomized end-time is required, the LCE is first moved to the 'Cancelled' list and the event is generated with randomized end-time specified. When the randomized end-time has been reached, the LCE is moved to the 'Deallocated' list and the event is 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 41.4.2). 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 41.4.2). 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 41.4.2) 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.

41.7.2 Other Elements of tsSE_DRLCCallBackMessage

In addition to the fields `eEventType` and `u8CommandId` described in Section 41.7.1, 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 41.9.

```
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 41.7.1).

41.8 Functions

The following DRLC cluster functions are provided:

41.8.1 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 creates 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 D.

Note: *This function must not be called for an endpoint on which a standard ZigBee device (example, 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 the ZigBee Devices User Guide*

Note: *(JNUG3131).*

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 ZCL has been initialized.

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 DRLC cluster.

The function initializes 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 Section 6.1.16). This structure is updated by the function by initializing individual structure fields.
- *psClusterDefinition*Pointer to structure indicating the type of cluster to be created (see Section 6.1.2). 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 is 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 initializes the attributes with default values.

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_INVALID_VALUE

41.8.2 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 is sent
- *u8DestinationEndPointId*: Number of the remote endpoint to which the LCE is 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 is sent
- *psLoadControlEvent*: Pointer to a structure (see Section 41.11.1) 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 41.9 or ZCL return code listed in Section 7.2

41.8.3 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 41.5.2.2 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 is sent
u8DestinationEndPointId: Number of the remote endpoint to which the request is 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 is sent (this must be the address of the ESP)
psGetScheduledEvents: Pointer to a structure which contains the LCE requirements of the request (see Section 41.11.2)
pu8TransactionSequenceNumber: Pointer to a location to store the Transaction Sequence Number (TSN) of the request

Returns

Any relevant DRLC return code listed in Section 41.9 or ZCL return code listed in Section 7.2

41.8.4 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 is cancelled locally and the cancellation is also sent to the specified DRLC cluster client endpoints. The LCE is 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 randomized end (for a full description of the end-time options, refer to Section 41.5.3).

Parameters

- *u8SourceEndPointId*: Number of the local endpoint through which the request is sent
- *u8DestinationEndPointId*: Number of the remote endpoint to which the request is 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 is sent
- *psCancelLoadControlEvent*: Pointer to a structure which contains the LCE cancellation requirements (see Section 41.11.3)
- *pu8TransactionSequenceNumber*: Pointer to a location to store the Transaction Sequence Number (TSN) of the request

Returns

Any relevant DRLC return code listed in Section 41.9 or ZCL return code listed in Section 7.2

41.8.5 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 are cancelled locally and the cancellation is also sent to the specified DRLC cluster client endpoints. The LCEs are 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 randomized end (for a full description of the end-time options, refer to Section 41.5.3).

Parameters

- *u8SourceEndPointId*: Number of the local endpoint through which the request is sent
- *u8DestinationEndPointId*: Number of the remote endpoint to which the request is 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 is sent
- *eCancelEventControl*: Enumeration indicating an immediate or randomized 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 41.9 or ZCL return code listed in Section 7.2

41.8.6 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 41.9 or ZCL return code listed in Section 7.2

41.8.7 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 41.11.1.

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 41.11.1)

Returns

Any relevant DRLC return code listed in Section 41.9 or ZCL return code listed in Section 7.2

41.8.8 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 41.11.1)

Returns

Any relevant DRLC return code listed in Section 41.9 or ZCL return code listed in Section 7.2

41.8.9 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 41.11.1)
- *peEventList*: Pointer to variable to receive enumerated value of the list in which the LCE was found (see Section 41.10.7)

Returns

Any relevant DRLC return code listed in Section 41.9 or ZCL return code listed in Section 7.2

41.9 Return codes

In addition to some of the ZCL status enumerations (detailed in Section 7.2), the following enumerations are returned by the DRLC cluster functions (described in Section 41.8) 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.

Table 77. Return codes

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 cancelation 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 recognized
E_SE_DRLC_BAD_CRITICALITY_LEVEL	Specified criticaility level not recognized
E_SE_DRLC_DURATION_TOO_LONG	Specified duration exceeds maximum of 1440 minutes (one day)

41.10 Enumerations

41.10.1 '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.

Table 78. Device Classes

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
E_SE_DRLC_GENERATION_SYSTEMS_BIT	Generation systems
E_SE_DRLC_DEVICE_CLASS_FIRST_RESERVED_BIT	Reserved

41.10.2 '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.

Table 79. DRLC Event Types

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

DRLC events are described in more detail in Section 41.7.

41.10.3 '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.

Table 80. Criticality Levels

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
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

41.10.4 'LCE Cancellation' Enumerations

The cancelation options (immediate or randomized) 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.

Table 81. LCE Cancelation Options

LCE Cancellation Enumeration	Description
E_SE_DRLC_CANCEL_CONTROL_IMMEDIATE	LCE is cancelled immediately by moving it directly to the 'Deallocated' list - a randomized end-time configured in the LCE is 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 is 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 41.2)

41.10.5 '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.

Table 82. LCE Participation Options

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)

41.10.6 '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 41.11.1).

Table 83. LCE Data Modification Options

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

Table 83. LCE Data Modification Options*...continued*

LCE Participation Enumeration	Description
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

41.10.7 '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.

Table 84. LCE Lists

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 randomized 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

41.10.8 '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.

Table 85. LCE Status Codes

Enumeration	Description
E_SE_DRLC_LOAD_CONTROL_EVENT_COM-MAND_ 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_PART ICIPATION	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)
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_COM-MAND_ REJECTED	LCE command rejected

41.11 Structures

41.11.1 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 41.10.3.
- `u8CoolingTemperatureOffset` (optional) specifies the required temperature offset, in units of 0.1°C, above the current temperature set-point of a cooling device (example, 0x5 represents a temperature offset of 0.5°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°C, below the current temperature set-point of a heating device (example, 0x14 represents a temperature offset of 2.0°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 randomize 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 41.10.1.
- `u16DurationInMinutes` specifies the duration, in minutes, of the LCE (although the actual duration will be longer than the specified duration if a randomized 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°C, for a cooling device, where a negative temperature is represented in two’s complement form (e.g. a temperature of 20°C is represented by 0x07D0 and -40°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°C, for a heating device, where a negative temperature is represented in two’s complement form (for example, a temperature of 25°C is represented by 0x09C4 and -1°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 is later if a randomized start-time is required). The value 0x000000000 is used to indicate a 'start-time of now'.

41.11.2 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

41.11.3 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 identifier (provided by the utility company) of the LCE to be canceled
- u16DeviceClass is a bitmap indicating the device class(es) to which the LCE cancelation applies - enumerations for the device classes are provided, as described in Section 41.10.1
- u8UtilityEnrolmentGroup is the enrolment group of the devices to which the LCE cancelation applies
- eCancelControl indicates whether to honour a randomised end that has been configured in the LCE - enumerations are provided, as described in Section 41.10.4
- u32effectiveTime is the time (UTC) from which the LCE cancelation is effective

41.11.4 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 41.10.8
- `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 41.11.1 (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 41.11.1 (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 41.10.3
- `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 41.11.1 (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 41.11.1 (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.*

41.11.5 tsSE_DRLCCallBackMessage

The structure of type `tsSE_DRLCCallBackMessage` contains a DRLC callback event. It is shown below but described in Section 41.7.

```
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;
```

41.12 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 41.4.2) 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 41.6), 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.

42 Simple Metering Cluster

This chapter outlines the Simple Metering cluster, which 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.

42.1 Overview

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

Table 86. Simple Metering Cluster in ZigBee Devices

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

Thus, a Metering Device or ESP can use this cluster to store attributes and respond to commands relating to these attributes. 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. Further compile-time options for the Simple Metering cluster are detailed in Section 42.12.

The information that can potentially be stored in this cluster is organized 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 42.3.

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 42.2).*

42.2 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:

42.2.1 '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
```

42.2.2 '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
```

42.2.3 '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 42.10.2 *(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
```

42.2.4 '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 42.10.3

- 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 42.10.4
- 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 42.10.6
- 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
```

42.2.5 '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
```

42.2.6 '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*.

42.2.7 'Supply Limit' Attribute Set

- The following are optional attributes and are fully described in the *ZigBee Smart Energy Profile Specification*:

```
u24CurrentDemandDelivered
u24DemandLimit
u8DemandIntegrationPeriod
u8NumberOfDemandSubintervals
```

42.2.8 '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_CU
RRENT_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]
```

42.3 Attribute Settings

The Simple Metering cluster contains both mandatory and optional attributes (see Section 42.2). 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 42.10.6.

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 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 42.10.2.

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 42.10.3.

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 to accommodate the above format information - these macros are detailed in Section 42.10.4.

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.

42.4 Remotely Reading Simple Metering Attributes

Dedicated functions are provided 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 2.3.2.

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 re-sends 'read attributes' requests until all attribute values are obtained.

Note that read access to cluster attributes must be explicitly enabled at compile-time as described in Section 1.3.

42.5 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 6 below.

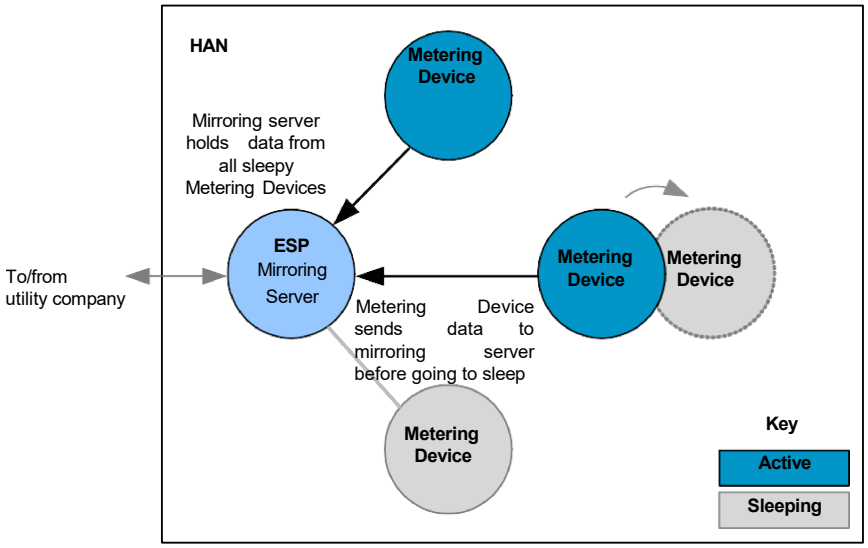


Figure 5. Mirroring of Metering Data

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 42.12). 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 7 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` contains a section on mirroring support which includes an array of `tsSE_Mirror` structures (see Section 42.11.2). 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.

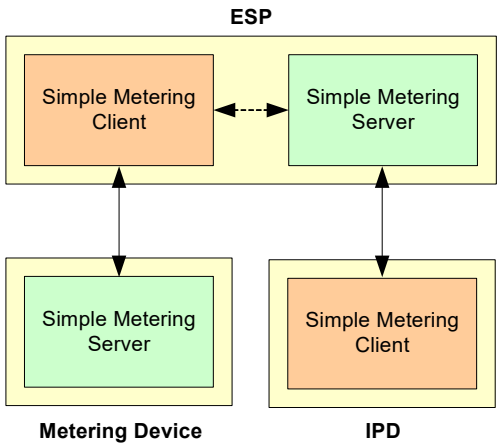


Figure 6. Simple Metering Cluster in Mirroring

42.5.1 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 42.12 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 are 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 are 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 are 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 using the NVM 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 42.5.2 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 NVM 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 42.5.3.

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;
     NvSaveOnIdle(&sMirrorState, TRUE);
}
break;
```

Recreating Mirrors Following an ESP Reset

If the ESP is reset, the mirrors that have been created on the device are 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.*

42.5.2 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 42.12 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 42.5.1.

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 42.11.6) 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 NVM module.
- The Metering Device application can now send metering data for storage on the ESP whenever required, as described in Section 42.5.3.

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.

42.5.3 Mirroring Data

Once a mirror for a Metering Device has been set up, as described in Section 42.5.1 and Section 42.5.2, 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 2.3.2.
- 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 Section 5.2.

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.

42.5.4 Reading Mirrored Data

A ZigBee 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 2.3.2 - 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.

42.5.5 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 NVM module. This step is illustrated in the code fragment under "Writing and Preserving Array of IEEE Addresses" on page 1011.

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.*

42.6 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 42.6.1).

If the 'Get Profile' feature is required, it must be enabled in the compile-time options as described in Section 42.12. These options include the maximum number of consumption intervals that can be archived on the server (and therefore requested).

42.6.1 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 42.12). 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 Section 18.7.*

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 42.2). Standard periods, ranging from 2.5 minutes to one day, are provided as a set of enumerations (see Section 42.10.10).

42.6.2 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 42.6.1)
- 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.

42.7 Simple Metering Events

The Simple Metering cluster has its own events that are handled through the callback mechanism described in Chapter 3. 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.

42.7.1 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 42.10.7) 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 42.7.2.

42.7.2 Command Types

For each event type listed in Section 42.7.1, 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

u8CommandId Enumeration	uMessage Union Element
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 42.10.8.

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 42.10.9.

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*.

42.8 Functions

The following Simple Metering cluster functions are provided:

1. eSE_SMCreate
2. eSE_ReadMeterAttributes
3. eSE_HandleReadMeterAttributesResponse
4. eSM_ServerRequestMirrorCommand
5. eSM_ServerRemoveMirrorCommand
6. eSM_CreateMirror
7. eSM_RemoveMirror
8. eSM_GetFreeMirrorEndPoint
9. eSM_IsMirrorSourceAddressValid
10. eSM_ServerUpdateConsumption
11. eSM_ClientGetProfileCommand
12. u32SM_GetReceivedProfileData

42.8.1 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 D.

Note: *This function must not be called for an endpoint on which a standard ZigBee device (for example, an IPD) is 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 the ZigBee Devices User Guide*

Note: *(JNUG3131).*

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 ZCL has been initialized.

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 SM_NUM_OF_ATTRIBUTES.

The array declaration should be as follows:

```
uint8 au8AppSMClusterAttributeControlBits[SM_NUM_OF_ATTRIBUTES];
```

The function initializes 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 Section 6.1.16). This structure is updated by the function by initialising individual structure fields.

- *psClusterDefinition*: Pointer to structure indicating the type of cluster to be created (see Section 6.1.2). 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 initialize the attributes with default values.

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_INVALID_VALUE

42.8.2 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 1.3.

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 are 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 is 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 is sent

- *u8DestinationEndPointId*: Number of the remote endpoint to which the request is 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 is 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

42.8.3 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 are 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. 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

42.8.4 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 42.5.

Parameters

- *u8SourceEndpoint*: Number of local endpoint through which request is sent
- *u8DestinationEndpoint*: Number of ESP endpoint to which request is sent (main endpoint of ESP)
- *psDestinationAddress*: Pointer to a structure containing the address of the ESP device (to which the request is 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

42.8.5 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 42.5.5.

Parameters

- *u8SourceEndpoint*: Number of local endpoint through which request is 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 is 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

42.8.6 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

42.8.7 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

42.8.8 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 42.5.1.

Parameters

- *pu16FreeEP*: Pointer to location to receive next free endpoint number

Returns

- E_ZCL_SUCCESS

42.8.9 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 42.5.3.

Parameters

- *psZCL_ReportAttributeMirror*: Pointer to `sReportAttributeMirror` element of the event

Returns

- E_ZCL_SUCCESS

42.8.10 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 42.11.5.

The buffer can contain a limited number of entries, determined at compile-time (see Section 42.12), 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 42.2). Standard periods, ranging from 2.5 minutes to one day, are provided as a set of enumerations (see Section 42.10.10).

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

42.8.11 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 is sent
- *u8DestinationEndpoint*: Number of endpoint to which request is 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

42.8.12 u32SM_GetReceivedProfileData

```
uint32 u32SM_GetReceivedProfileData(
        tsSM_GetProfileResponseCommand
 *psSMGetProfileResponseCommand);
```

Description

This function is used on a Simple Metering cluster client to obtain the consumption data received in a ‘Get Profile’ response. The response is obtained 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. This response causes 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 returns 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 returns 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.

42.9 Return codes

The Simple Metering cluster functions use the ZCL return codes, listed in Section 7.2.

42.10 Enumerations

42.10.1 ‘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 42.2).*

```
typedef enum PACK
{
    /* Reading information attribute set attribute IDs*/
    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 set attribute IDs */
    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 IDs */
    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 IDs */
    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 IDs */
    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 IDs */
    E_CLD_SM_ATTR_ID_MAX_NUMBER_OF_PERIODS_DELIVERED     =    0x0500,
    /* Supply Limit attribute set attribute IDs */
    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 IDs */
    /* 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 IDs */
    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;
```

42.10.2 '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:

Table 87. 'Meter Status' Enumerated Masks

Enumeration	Description
E_CLD_SM_METER_STATUS_CHECK_METER_MASK	Non-fatal problem detected on meter

Table 87. 'Meter Status' Enumerated Masks*...continued*

Enumeration	Description
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

42.10.3 '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.

Table 88. Units of Measure Enumerations

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)

Table 88. Units of Measure Enumerations*...continued*

Enumeration	Description	
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	-

42.10.4 '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:

Table 89. 'Summation Formatting' Enumerations

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

Table 89. 'Summation Formatting' Enumerations*...continued*

Enumeration	Description
E_CLD_SM_FORMATTING_SUPPRESS_LEADING_ZEROS_BIT	Bit-mask used to extract bit indicating whether lead-ing zeros will be suppressed

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)
```

42.10.5 '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.

Table 90. 'Supply Direction' Enumerations

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

42.10.6 '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.

Table 91. 'Metering Device Type' Enumerations

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

42.10.7 '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.

Table 92. Simple Metering Event Types

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

Table 92. Simple Metering Event Types*...continued*

Event Type Enumeration	Description
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*)

42.10.8 '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;
```

Table 93. Commands Issued by Server

Command Enumeration	Description
E_CLD_SM_GET_PROFILE_RESPONSE	Response to 'Get Profile' request - content of response is contained in the structure `tsS-M_GetProfileResponse Command` in the event (see Section 42.11.9)
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 42.11.10)

42.10.9 '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;
```

Table 94. Commands Issued by Client

Command Enumeration	Description
E_CLD_SM_GET_PROFILE	A 'Get Profile' request - content of request is contained in the structure `ts SM_GetProfile-RequestCommand` in the event (see Section 42.11.8)
E_CLD_SM_REQUEST_MIRROR_ RESPONSE	Response to 'Add Mirror' request - content of response is contained in the structure `tsS-M_RequestMirrorResponseCommand` in the event (see Section 42.11.6)

Table 94. Commands Issued by Client*...continued*

Command Enumeration	Description
E_CLD_SM_MIRROR_REMOVED	Response to 'Remove Mirror' request - content of response is contained in the structure `tsS-M_MirrorRemovedResponseCommand` in the event (see Section 42.11.7)
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 42.11.10)

42.10.10 '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;
```

Table 95. 'Consumption Interval' Enumerations

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

42.10.11 '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;
```

Table 96. Status Enumerations

Status Enumeration	Description
E_CLD_SM_STATUS_SUCCESS	Success
E_CLD_SM_STATUS_UNDEFINED_INTERVAL_CHANNEL	Undefined `eIntervalChannel` value speci-fied in 'Get Profile' request (see Section 42.11.8)
E_CLD_SM_STATUS_INTERVAL_NOT_SUPPORTED	Unsupported consumption data specifed through `eIntervalChannel` in 'Get Profile' request (see Section 42.11.8)
E_CLD_SM_STATUS_INVALID_END_TIME	Invalid end-time specified in 'Get Profile' request (Section 42.11.8)
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

42.11 Structures

42.11.1 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 42.10.7
- `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 42.10.8

  - E_CLD_SM_SERVER_RECEIVED_COMMAND - enumerated commands are provided, as described in Section 42.10.9
- `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 42.11.9
  - `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 42.11.8
  - `sRequestMirrorResponseCommand` is a structure containing the payload of an 'Add Mirror' response - see Section 42.11.6
  - `sMirrorRemovedResponseCommand` is a structure containing the payload of an 'Remove Mirror' response - see Section 42.11.7
  - `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 42.11.10

42.11.2 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. The `tsSE_Mirror` structure is shown and described below.

Note: *This structure is only for use by the ZCL and should not be modified by the application.*

```
typedef struct
{
            /*Mirrored EndPoint*/
    tsZCL_EndPointDefinition sEndPoint;
    /*Mirror Requester address*/
    uint64        u64SourceAddress;
    /*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 details of this structure, refer to Section 6.1.1)
- `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 42.11.3) 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 42.11.4) which contains data relating to a received command/message for the mirror endpoint

42.11.3 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 ZCL 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 details of this structure, refer to Section 6.1.16)
- `sSM_Cluster` is a `tsZCL_ClusterInstance` structure which contains information on the Simple Metering cluster instance associated with a mirror endpoint (for details of this structure, refer to Section 6.1.16)

42.11.4 tsSM_CustomStruct

This structure contains data relating to a command/message for a mirror endpoint.

Note: *This structure is only for use by the ZCL 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
- `sSMCustomCallBackEvent` is a `tsZCL_CallBackEvent` structure (see Section 3.1) which contains the event that has been generated as a result of the received command/message
- `sSMCallBackMessage` is a `tsSM_CallBackMessage` structure (see Section 42.11.1) which contains details of the event and the command/message that caused the event

42.11.5 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

42.11.6 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.

42.11.7 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.

42.11.8 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 Section 6.1.4)

42.11.9 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 42.10.11
- 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 42.10.10
- u8NumberOfPeriodsDelivered is the number of consumption intervals being reported
- pau24Intervals is a pointer to the consumption data being reported

42.11.10 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

42.12 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 42.5), 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 are 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 are 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 42.6), 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
#define      CLD_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).

Part X: Commissioning Clusters

This part comprises two chapters:

- Chapter 43 details the **Commissioning** cluster
- Chapter 44 details the **Touchlink Commissioning** cluster

43 Commissioning Cluster

This chapter details the Commissioning cluster which is defined in the ZCL and is a optional cluster for all ZigBee devices.

The Commissioning cluster has a Cluster ID of 0x0015.

43.1 Overview

The Commissioning cluster is 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 server must be implemented on a device that is to be commissioned into a network.
- The Commissioning cluster client must be implemented on a device that can initiate the commissioning of another device into a network - for example, on a commissioning tool.

This optional cluster is enabled by defining CLD_COMMISSIONING in the **zcl_options.h** file. The inclusion of the client or server software must also be pre-defined in the application's compile-time options (in addition, if the cluster is to reside on a custom endpoint then the role of client or server must also be specified when creating the cluster instance). The compile-time options for the Commissioning cluster are fully detailed in Section 43.10.

Only server attributes are supported and all are optional - the required attributes must be enabled in the compile-time options. The information that can potentially be stored in the Commissioning cluster is organised into the following attribute sets: Start-up Parameters, Join Parameters, End Device Parameters, Concentrator Parameters.

Note: *The attribute values are set by the application but the application must ensure that these values are synchronized with the settings and NIB values for the ZigBee PRO stack.*

The Commissioning cluster also provides optional commands, which can be enabled in the compile-time options.

43.2 Commissioning Cluster structure and attributes

The Commissioning cluster has only server attributes that are contained in the following `tsCLD_Commissioning` structure, organised as a set of structures containing the Commissioning cluster attribute sets:

```
typedef struct
{
#ifdef COMMISSIONING_SERVER
/* Start- attribute setup Parameters attribute set */
    tsCLD_StartupParameters         sStartupParameters;
/* Join Parameters attribute set */
    tsCLD_JoinParameters            sJoinParameters;
/* End Device Parameters attribute set */
    tsCLD_EndDeviceParameters       sEndDeviceParameters;
/* Concentrator Parameters attribute set */
    tsCLD_ConcentratorParameters    sConcentratorParameters;
#endif
    zuint16       u16ClusterRevision;
} tsCLD_Commissioning;
```

where:

```
sStartupParameters is a structure containing the attributes of the Start-
up Parameters attribute set - this structure and the associated attributes are detailed in
sJoinParameters is a structure containing the attributes of the Join Parameters attribute
sEndDeviceParameters is a structure containing the attributes of the End Device Parameters
sConcentratorParameters is a structure containing the attributes of the Concentrator Param
u16ClusterRevision is a mandatory attribute that specifies the revision of the cluster spe
```

Note: *Memory is allocated at compile-time for all the Commissioning cluster attributes.*

43.2.1 Start-up Parameters (tsCLD_StartupParameters)

The `tsCLD_StartupParameters` structure below contains the attributes of the Start-up Parameters attribute set:

```
typedef struct
{
#ifdef     CLD_COMMISSIONING_ATTR_SHORT_ADDRESS
    uint16              u16ShortAddress;
#endif
#ifdef     CLD_COMMISSIONING_ATTR_EXTENED_PAN_ID
    zieeeaddress        u64ExtPanId;
#endif
#ifdef     CLD_COMMISSIONING_ATTR_PAN_ID
    uint16              u16PANId;
#endif
#ifdef     CLD_COMMISSIONING_ATTR_CHANNEL_MASK
    zbmap32             u32ChannelMask;
#endif
#ifdef     CLD_COMMISSIONING_ATTR_PROTOCOL_VERSION
    uint8               u8ProtocolVersion;
#endif
#ifdef     CLD_COMMISSIONING_ATTR_STACK_PROFILE
    uint8               u8StackProfile;
#endif
#ifdef     CLD_COMMISSIONING_ATTR_START_UP_CONTROL
    zenum8               e8StartUpControl;
#endif
#ifdef     CLD_COMMISSIONING_ATTR_TC_ADDR
    zieeeaddress         u64TcAddr;
#endif
#ifdef     CLD_COMMISSIONING_ATTR_TC_MASTER_KEY
    tsZCL_Key            sTcMasterKey;
#endif
#ifdef     CLD_COMMISSIONING_ATTR_NWK_KEY
    tsZCL_Key           sNwkKey;
#endif
#ifdef     CLD_COMMISSIONING_ATTR_USE_INSECURE_JOIN
    bool_t              bUseInsecureJoin;
#endif
#ifdef     CLD_COMMISSIONING_ATTR_PRE_CONFIG_LINK_KEY
    tsZCL_Key           sPreConfigLinkKey;
#endif
#ifdef     CLD_COMMISSIONING_ATTR_NWK_KEY_SEQ_NO
    uint8               u8NwkKeySeqNo;
#endif
#ifdef     CLD_COMMISSIONING_ATTR_NWK_KEY_TYPE
    zenum8              e8NwkKeyType;
```

```
#endif
#ifdef     CLD_COMMISSIONING_ATTR_NWK_MANAGER_ADDR
    uint16              u16NwkManagerAddr;
#endif
} tsCLD_StartupParameters;
```

where:

- `u16ShortAddress` is the intended 16-bit network address of the device (which will be used provided that the address is not to be obtained from the parent - that is, on the Co-ordinator or on other ZigBee PRO devices for which `e8StartUpControl` is set to 0x00).
- `u64ExtPanId` is the 64-bit Extended PAN ID of the network which the device should join (the special value of 0xFFFFFFFF can be used to specify no particular network).
- `u16PANId` is the 16-bit PAN ID of the network which the device should join (which will be used provided that the PAN ID is not to be obtained from the parent - that is, on the Co-ordinator or on other ZigBee PRO devices for which `e8StartUpControl` is set to 0x00).
- `u32ChannelMask` is a 32-bit bitmap representing an IEEE 802.15.4 channel mask which indicates the set of radio channels that the device should scan as part of the network join or formation process.
- `u8ProtocolVersion` is used to indicate the ZigBee protocol version that the device is to support (only needed if the device potentially supports multiple versions).
- `u8StackProfile` is used to indicate the stack profile to be implemented on the device - the possible values are 0x01 for ZigBee Stack profile and 0x02 for ZigBee PRO Stack profile.
- `e8StartUpControl` is an enumeration which is used to indicate the start-up mode of the device (e.g. device should form a network with the specified Extended PAN ID) and therefore determines how certain other attributes will be used. For further information on how this attribute is used, refer to the ZCL Specification.
- `u64TcAddr` is the 64-bit IEEE/MAC address of the Trust Centre node for the network with the specified Extended PAN ID (this is needed if security is to be implemented).
- `sTcMasterKey` is the master key to be used during key establishment with the specified Trust Centre (this is needed if security is to be implemented). The default is a 128-bit zero value indicating that the key is unspecified.
- `sNwkKey` is the network key to be used when communicating within the network with the specified Extended PAN ID (this is needed if security is to be implemented). The default is a 128-bit zero value indicating that the key is unspecified.
- `bUseInsecureJoin` is a Boolean flag which, when set to TRUE, allows an unsecured join as a fall-back (even if security is enabled).
- `sPreConfigLinkKey` is the pre-configured link key between the device and the Trust Centre (this is needed if security is to be implemented). The default is a 128-bit zero value indicating that the key is unspecified.
- `u8NwkKeySeqNo` is the 8-bit sequence number for the network key. The default value is 0x00.
- `e8NwkKeyType` is the type of the network key. The default value is 0x01 when `u8StackProfile` is 0x01 and 0x05 when `u8StackProfile` is 0x02.
- `u16NwkManagerAddr` is the 16-bit network address of the Network Manager. The default value is 0x0000, indicating that the Network Manager is the ZigBee Co-ordinator.

43.2.2 Join Parameters (tsCLD_JoinParameters)

The `tsCLD_JoinParameters` structure below contains the attributes of the Join Parameters attribute set:

```
typedef struct
{
#ifdef     CLD_COMMISSIONING_ATTR_SCAN_ATTEMPTS
    uint8          u8ScanAttempts;
#endif
#ifdef     CLD_COMMISSIONING_ATTR_TIME_BW_SCANS
```

```
    uint16          u16TimeBwScans;
#endif
#ifdef     CLD_COMMISSIONING_ATTR_REJOIN_INTERVAL
    uint16          u16RejoinInterval;
#endif
#ifdef     CLD_COMMISSIONING_ATTR_MAX_REJOIN_INTERVAL
    uint16          u16MaxRejoinInterval;
#endif
} tsCLD_JoinParameters;
```

where:

- `u8ScanAttempts` is the number of scan attempts to make before selecting a parent to join. The default value is 0x05.
- `u16TimeBwScans` is the time-interval, in milliseconds, between consecutive scan attempts. The default value is 0x64.
- `u16RejoinInterval` is the time-interval, in seconds, between consecutive attempts to rejoin the network for an End Device which has lost its network connection. The default value is 0x3C.
- `u16MaxRejoinInterval` is an upper limit, in seconds, on the value of the `u16RejoinInterval` attribute. The default value is 0x0E10.

43.2.3 End Device Parameters (tsCLD_EndDeviceParameters)

The `tsCLD_EndDeviceParameters` structure below contains the attributes of the End Device Parameters attribute set:

```
typedef struct
{
#ifdef     CLD_COMMISSIONING_ATTR_INDIRECT_POLL_RATE
    uint16          u16IndirectPollRate;
#endif
#ifdef     CLD_COMMISSIONING_ATTR_PARENT_RETRY_THRSHLD
    uint8           u8ParentRetryThreshold;
#endif
} tsCLD_EndDeviceParameters;
```

where:

- `u16IndirectPollRate` is the time-interval, in milliseconds, between consecutive polls from an End Device which polls its parent while awake (an End Device with a receiver that is inactive while sleeping).
- `u8ParentRetryThreshold` is the number of times that an End Device should attempt to re-contact its parent before initiating the rejoin process.

43.2.4 Concentrator Parameters (tsCLD_ConcentratorParameters)

The `sCLD_ConcentratorParameters` structure below contains the attributes of the Concentrator Parameters attribute set:

```
typedef struct
{
#ifdef     CLD_COMMISSIONING_ATTR_CONCENTRATOR_FLAG
    bool_t          bConcentratorFlag;
#endif
#ifdef     CLD_COMMISSIONING_ATTR_CONCENTRATOR_RADIUS
    uint8         u8ConcentratorRadius;
#endif
#ifdef     CLD_COMMISSIONING_ATTR_CONCENTRATOR_DISCVRY_TIME
```

```
    uint8          u8ConcentratorDiscoveryTime;
#endif
} tsCLD_ConcentratorParameters;
```

where:

- `bConcentratorFlag` is a Boolean flag which, when set to TRUE, enables the device as a concentrator for many-to-one routing. The default value is FALSE.
- `u8ConcentratorRadius` is the hop-count radius for concentrator route discoveries. The default value is 0x0F.
- `u8ConcentratorDiscoveryTime` is the time-interval, in seconds, between consecutive discoveries of inbound routes initiated by the concentrator. The default value is 0x0000, indicating that this time-interval is unknown and the discoveries must be triggered by the application.

43.3 Attribute Settings

The Commissioning cluster structure contains only optional attributes. Each attribute is enabled/disabled through a corresponding macro defined in the **zcl_options.h** file (see Section 43.10) - for example, `u16ShortAddress` is enabled/disabled through the macro CLD_COMM_ATTR_SHORT_ADDRESS.

The function **eCLD_CommissioningSetAttribute()** can be used on the cluster server to write values to any one of the four attribute sets of the Commissioning cluster.

43.4 Initialisation

The function **eCLD_CommissioningClusterCreateCommissioning()** is used to create an instance of the Commissioning cluster. The function is generally called by the initialization function for the host device.

43.5 Commissioning Commands

A number of commissioning commands are provided to allow a Commissioning cluster client to remotely request actions relating to the Start-up Parameters attribute set (see Section 43.2.1) on a cluster server. This includes initiating a device restart from the current Start-up Parameter values, as well as the management of these attributes.

43.5.1 Device Start-up

The ‘current’ set of Start-up Parameter values on a cluster server are those used in the start-up procedure, which can be remotely initiated from a cluster client using the function **eCLD_CommissioningCommand RestartDeviceSend()**. This function sends a Restart Device command to the remote device hosting the cluster server. This command provides a number of options concerning the timing of the restart:

- **Without delay:** The start-up procedure is invoked as soon as the command is received. This option requires both the delay and jitter to be specified as zero.
- **With delay:** The start-up procedure is invoked after a specified delay (in seconds). If no delay is required, the delay period must be specified as zero.
- **With delay and jitter:** The start-up procedure is invoked after a specified delay (in seconds) with a random jitter period added. It is necessary to indicate a maximum jitter period but the actual period will be randomly generated. If no jitter is required, the maximum jitter period must be specified as zero.

Note: *If only jitter is required, the delay period must be specified as zero and the maximum jitter period must be non-zero.*

In all of the above cases, it is possible to configure the start-up procedure to begin either without any further delay or at a 'convenient' moment (when there are no pending actions that should be completed before the restart).

The above options are configured in the command payload (see Section 43.9.2).

The cluster server will send a Restart Device Response to the requesting client before invoking the start-up procedure or starting the countdown (for the delay).

43.5.2 Stored Start-up Parameters

In addition to the 'current' set of values for the Start-up Parameters attribute set, the cluster server can store other sets of values for these attributes in non-volatile memory. Each stored set of Start-up Parameter values is assigned a unique index number. At any time, a particular stored set of values can be retrieved and loaded to become the current set. Functions are provided for managing the saved sets of Start-up Parameter values.

43.5.2.1 Saving Start-up Parameters

In order to save a set of Start-up Parameter values, it is first necessary to set them as the current attribute values - this must be done locally by the application on the device hosting the server, possibly using the function **eCLD_CommissioningSetAttribute()**.

The application on a device hosting a cluster client can send a Save Start-up Parameters command to the cluster server in order to request that the current set of Start-up Parameter values is saved to non-volatile memory. This can be done by calling **eCLD_CommissioningCommandSaveStartupParamsSend ()** or, alternatively, **eCLD_CommissioningCommandModifyStartupParamsSend ()**. The index number of the saved record must be specified in the request. If this number has already been used, the existing stored values will be over-written with the new values.

It is the responsibility of the user application on the device hosting the server to perform the save. When the command arrives, a ZCL custom event will be generated and the request should be handled by the user-defined callback function for the endpoint on which the application is located. The server will automatically send a Save Start-up Parameters Response to the requesting client.

43.5.2.2 Retrieving Stored Start-up Parameters

A set of Start-up Parameter values that have been stored by in non-volatile memory (as described in Section 43.5.2.1) can be retrieved and loaded as the current set of values. The required stored set of values is specified using its unique index number.

The application on a device hosting a cluster client can send a Restore Start-up Parameters command to the cluster server in order to request that the specified set of Start-up Parameter values is loaded from non-volatile memory. This can be done by calling **eCLD_CommissioningCommandRestoreStartupParamsSend ()** or, alternatively, **eCLD_CommissioningCommandModifyStartupParamsSend ()**. The index number of the relevant set must be specified in the request.

It is the responsibility of the user application on the device hosting the server to retrieve the relevant set of values and load them as the current values. When the command arrives, a ZCL custom event will be generated and the request should be handled by the user-defined callback function for the endpoint on which the application is located. The server will automatically send a Restore Start-up Parameters Response to the requesting client.

A device restart is required in order to implement the loaded values, as described in Section 43.5.1.

43.5.3 Reset Start-up Parameters to Default Values

A set of Start-up Parameters on the cluster server can be reset to their default values.

The application on a device hosting a cluster client can send a Reset Start-up Parameters command to the cluster server in order to request that the Start-up Parameters are reset to their default values. This can be done by calling **eCLD_CommissioningCommandResetStartupParamsSend ()** or, alternatively, **eCLD_CommissioningCommandModifyStartupParamsSend ()**. Options are available concerning the set(s) of Start-up Parameters to reset - any combination of the following can be performed:

- Reset the current set of Start-up Parameters
- Reset all stored sets of Start-up Parameters or the stored set with given index
- Erase the stored set of Start-up Parameters with given index

The required options must be specified in the request. The option to erase a stored set of Start-up Parameters allows storage space to be freed up.

It is the responsibility of the user application on the device hosting the server to reset the relevant set(s) of values. When the command arrives, a ZCL custom event will be generated and the request should be handled by the user-defined callback function for the endpoint on which the application is located. The server will automatically send a Reset Start-up Parameters Response to the requesting client.

A device restart is required in order to implement the reset (current) values, as described in Section 43.5.1.

43.6 Commissioning Events

The Commissioning cluster has its own events that are handled through the callback mechanism outlined in Chapter 3. If a device uses this cluster then application-specific Commissioning event handling must be included in the user-defined callback function for the associated endpoint, where this callback function is registered through the relevant endpoint registration function. This callback function will then be invoked when a Commissioning event occurs and needs the attention of the application.

For a Commissioning 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 `tsCLD_CommissioningCallBackMessage` structure (fully detailed in Section 43.9.6).

```
typedef struct
{
    uint8                       u8CommandId;
    union
    {
        tsCLD_Commissioning_RestartDevicePayload
                                *psRestartDevicePayload;
        tsCLD_Commissioning_ModifyStartupParametersPayload
                                *psModifyStartupParamsPayload;
    } uReqMessage;
    union
    {
        tsCLD_Commissioning_ResponsePayload
                                *psCommissioningResponsePayload;
    } uRespMessage;
} tsCLD_CommissioningCallBackMessage;
```

When a Commissioning event occurs, one of a number of command types could have been received. The relevant command type is specified through the `u8CommandId` field of the `tsCLD_CommissioningCallBackMessage` structure. The possible command types are detailed below.

The table below details the command types that can be received by the cluster server.

Table 97. Commissioning Command Types (on Server)

u8CommandId Enumeration	Description
E_CLD_COMMISSIONING_CMD_RESTART_DEVICE	A Restart Device command has been received
E_CLD_COMMISSIONING_CMD_SAVE_STARTUP_PARAMS	A Save Start-up Parameters command has been received
E_CLD_COMMISSIONING_CMD_RESTORE_STARTUP_PARAMS	A Restore Start-up Parameters command has been received
E_CLD_COMMISSIONING_CMD_RESET_STARTUP_PARAMS	A Reset Start-up Parameters command has been received

The table below details the command types that can be received by the cluster client.

Table 98. Commissioning Command Types (on Client)

u8CommandId Enumeration	Description
E_CLD_COMMISSIONING_CMD_RESTART_DEVICE	A Restart Device response has been received
E_CLD_COMMISSIONING_CMD_SAVE_STARTUP_PARAMS	A Save Start-up Parameters response has been received
E_CLD_COMMISSIONING_CMD_RESTORE_STARTUP_PARAMS	A Restore Start-up Parameters response has been received
E_CLD_COMMISSIONING_CMD_RESET_STARTUP_PARAMS	A Reset Start-up Parameters response has been received

43.7 Functions

The following Commissioning cluster function is provided:

1. eCLD_CommissioningClusterCreateCommissioning
2. eCLD_CommissioningCommandRestartDeviceSend
3. eCLD_CommissioningCommandSaveStartupParamsSend
4. eCLD_CommissioningCommandRestoreStartupParamsSend
5. eCLD_CommissioningCommandResetStartupParamsSend
6. eCLD_CommissioningCommandModifyStartupParamsSend
7. eCLD_CommissioningSetAttribute

43.7.1 eCLD_CommissioningClusterCreateCommissioning

```
teZCL_Status eCLD_CommissioningClusterCreateCommissioning(
        tsZCL_ClusterInstance *psClusterInstance,
        bool_t bIsServer,
        tsZCL_ClusterDefinition *psClusterDefinition,
        void *pvEndPointSharedStructPtr,
        uint8 *pu8AttributeControlBits,
        tsCLD_CommissioningCustomDataStructure
 *psCustomDataStructure);
```

Description

This function creates an instance of the Commissioning 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 Commissioning 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 D.

Note: *This function must not be called for an endpoint on which a standard ZigBee device is 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 the ZigBee Devices User Guide*

Note: *(JNUG3131).*

When used, this function must be called after the stack has been started and after the ZCL has been initialized.

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 is automatically adjusted by the compiler using the following declaration:

```
uint8 au8CommissioningAttributeControlBits
[(sizeof(asCLD_CommissioningClusterAttributeDefinitions) / sizeof(tsZCL_AttributeDefinitio
```

Parameters

psClusterInstance Pointer to structure containing information about the cluster instance to be created (see Section 6.1.16). This structure is updated by the function by initialising individual structure fields.

bIsServer Type of cluster instance (server or client) to be created:

TRUE - server

FALSE - client

psClusterDefinition Pointer to structure indicating the type of cluster to be created (see Section 6.1.2). In this case, this structure must contain the details of the Commissioning cluster. This parameter can refer to a pre-filled structure called `sCLD_Commissioning` which is provided in the **Commissioning.h** file.

pvEndPointSharedStructPtr Pointer to the shared structure used for attribute storage. This parameter should be the address of the structure of type `tsCLD_Commissioning` which defines the attributes of Commissioning cluster. The function initializes the attributes with default values.

pu8AttributeControlBits Pointer to an array of **uint8** values, with one element for each attribute in the cluster (see above). For a cluster client, set this pointer to NULL.

psCustomDataStructure Pointer to a structure containing the storage for internal functions of the cluster (see Section 43.9.5)

Returns

E_ZCL_SUCCESS
E_ZCL_FAIL
E_ZCL_ERR_PARAMETER_NULL
E_ZCL_ERR_INVALID_VALUE

43.7.2 eCLD_CommissioningCommandRestartDeviceSend

```
teZCL_Status eCLD_CommissioningCommandRestartDeviceSend(
        uint8 u8SourceEndPointId,
        uint8 u8DestinationEndPointId,
        tsZCL_Address *psDestinationAddress,
        uint8 *pu8TransactionSequenceNumber,                tsCLD_Commissioning_RestartDe
 *psPayload);
```

Description

This function can be used on a Commissioning cluster client to send a Restart Device command to a cluster server on a remote device. This command is used to run the start-up procedure with a new set of values for the Start-up Parameters attributes (these values must already be installed). The new values may be implemented immediately or after a specified delay with an optional jitter.

When the command arrives, a ZCL custom event will be generated and the request should be handled by the user-defined callback function for the endpoint on which the application is located (see Section 43.6). Before running the start-up procedure or starting the countdown (for the delay), the server will send a Restart Device Response to the requesting client, where a ZCL custom event will be generated.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is 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 to send the request. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values

u8DestinationEndPointId Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP

psDestinationAddress Pointer to a structure holding the address of the node to which the request is sent

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

psPayload Pointer to a structure containing the payload for this message (see Section 43.9.2)

Returns

E_ZCL_SUCCESS
E_ZCL_FAIL
E_ZCL_ERR_PARAMETER_NULL
E_ZCL_ERR_INVALID_VALUE

43.7.3 eCLD_CommissioningCommandSaveStartupParamsSend

```
teZCL_Status eCLD_CommissioningCommandSaveStartupParamsSend(
        uint8 u8SourceEndPointId,
        uint8 u8DestinationEndPointId,
        tsZCL_Address *psDestinationAddress,
        uint8 *pu8TransactionSequenceNumber,                  tsCLD_Commissioning_M
 *psPayload);
```

Description

This function can be used on a Commissioning cluster client to send a Save Start-up Parameters command to a cluster server on a remote device. This command instructs the server to locally save a set of values for the attributes of the Start-up Parameters attribute set. A device can store different sets of start-up parameters (in non-volatile memory), with each set being referenced using an index number. This index number must be specified and if a set has already been stored with the same index number then the stored values will be over-written with the new values.

It is the responsibility of the user application on the device hosting the server to implement the command. When the command arrives, a ZCL custom event will be generated and the request should be handled by the user-defined callback function for the endpoint on which the application is located (see Section 43.6). The server will automatically send a Save Start-up Parameters Response to the client, where a ZCL custom event will be generated.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is 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 to send the request. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values

u8DestinationEndPointId Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP

psDestinationAddress Pointer to a structure holding the address of the node to which the request is sent

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

psPayload Pointer to a structure containing the payload for this message (see Section 43.9.3)

Returns

E_ZCL_SUCCESS
E_ZCL_FAIL
E_ZCL_ERR_PARAMETER_NULL
E_ZCL_ERR_INVALID_VALUE

43.7.4 eCLD_CommissioningCommandRestoreStartupParamsSend

```
teZCL_Status eCLD_CommissioningCommandRestoreStartupParamsSend(
        uint8 u8SourceEndPointId,
        uint8 u8DestinationEndPointId,
        tsZCL_Address *psDestinationAddress,
        uint8 *pu8TransactionSequenceNumber,                                    tsCLD_Commiss
 *psPayload);
```

Description

This function can be used on a Commissioning cluster client to send a Restore Start-up Parameters command to a cluster server on a remote device. This command instructs the server to load a saved set of values for the attributes of the Start-up Parameters attribute set. The index of the required set of Start-up Parameters must be

specified in the command payload. Note that the command does not instruct the server to implement the loaded values using the start-up procedure - a Restart Device command is required to do this.

It is the responsibility of the user application on the device hosting the server to implement the command. When the command arrives, a ZCL custom event will be generated and the request should be handled by the user-defined callback function for the endpoint on which the application is located (see Section 43.6). The server will automatically send a Restore Start-up Parameters Response to the client, where a ZCL custom event will be generated.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is 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 to send the request. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values

u8DestinationEndPointId Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP

psDestinationAddress Pointer to a structure holding the address of the node to which the request is sent

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

psPayload Pointer to a structure containing the payload for this message (see Section 43.9.3)

Returns

E_ZCL_SUCCESS
E_ZCL_FAIL
E_ZCL_ERR_PARAMETER_NULL
E_ZCL_ERR_INVALID_VALUE

43.7.5 eCLD_CommissioningCommandResetStartupParamsSend

```
teZCL_Status eCLD_CommissioningCommandResetStartupParamsSend(
        uint8 u8SourceEndPointId,
        uint8 u8DestinationEndPointId,
        tsZCL_Address *psDestinationAddress,
        uint8 *pu8TransactionSequenceNumber,                    tsCLD
 *psPayload);
```

Description

This function can be used on a Commissioning cluster client to send a Reset Start-up Parameters command to a cluster server on a remote device. This command instructs the server to set the current Start-up Parameters to their default values. It is also possible to set one or all of any saved sets of Start-up Parameters to the defaults. The command can also be used to delete a specified set of saved Start-up Parameters.

It is the responsibility of the user application on the device hosting the server to implement the command. When the command arrives, a ZCL custom event will be generated and the request should be handled by the user-defined callback function for the endpoint on which the application is located (see Section 43.6). The server will automatically send a Reset Start-up Parameters Response to the client, where a ZCL custom event will be generated.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is 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 to send the request. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values

u8DestinationEndPointId Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP

psDestinationAddress Pointer to a structure holding the address of the node to which the request is sent

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

psPayload Pointer to a structure containing the payload for this message (see Section 43.9.3)

Returns

E_ZCL_SUCCESS
E_ZCL_FAIL
E_ZCL_ERR_PARAMETER_NULL
E_ZCL_ERR_INVALID_VALUE

43.7.6 eCLD_CommissioningCommandModifyStartupParamsSend

```
teZCL_Status eCLD_CommissioningCommandModifyStartupParamsSend(
        uint8 u8SourceEndPointId,
        uint8 u8DestinationEndPointId,
        tsZCL_Address *psDestinationAddress,
        uint8 *pu8TransactionSequenceNumber,
        tsCLD_Commissioning_ModifyStartupParametersPayload
 *psPayload,
        teCLD_Commissioning_Command
 eCLD_Commissioning_Command);
```

Description

This function can be used on a Commissioning cluster client to send a command to modify a set of values for the Start-up Parameters attributes in the cluster server on a remote device. One of four commands can be specified and sent, as listed and described in the table below:

Command	Description
Restart Device	Used to run the start-up procedure with the current set of values for the Start-up Parameters attributes, as described for the function **eCLD_Com-missioningCommandRestartDevice Send()**. These values may have been loaded using the Restore Start-up Parameters or Reset Start-up Parameters command.
Save Start-up Parameters	Used to save the current set of Start-up Parameter values with the speci-fied index, as described for the function **eCLD_CommissioningCom-mandSaveStartupParamsSend()**.
Restore Start-up Parameters	Used to load the saved set of Start-up Parameter values with the specified index, such that these values become the current Start-up Parameter val-ues, as described for the function **e**

Command	Description
	CLD_CommissioningCommandRe-storeStartupParamsSend(). Note that these values are not implemented, which requires a Restart Device command.
Reset Start-up Parameters	Used to reset the current Start-up Parameters to their defaults. One or all of any stored sets of Start-up Parameter values can also be reset to the defaults, as described for the function **e CLD_CommissioningComman-dResetStartupParamsSend()**. The command can also be used to delete a particular set of stored Start-up Parameters.

It is the responsibility of the user application on the device hosting the server to implement the command. When the command arrives, a ZCL custom event will be generated and the request should be handled by the user-defined callback function for the endpoint on which the application is located (see Section 43.6). The server will automatically send a response for the relevant command to the client, where a ZCL custom event will be generated.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is 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 to send the request. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values

u8DestinationEndPointId Number of the endpoint on the remote node to which the request is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP

psDestinationAddress Pointer to a structure holding the address of the node to which the request is sent

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

psPayload Pointer to a structure containing the payload for this message (see Section 43.9.3)

eCLD_Commissioning_Command Type of command to send, one of:
E_CLD_COMMISSIONING_CMD_RESTART_DEVICE
E_CLD_COMMISSIONING_CMD_SAVE_STARTUP_PARAMS
E_CLD_COMMISSIONING_CMD_RESTORE_STARTUP_PARAMS
E_CLD_COMMISSIONING_CMD_RESET_STARTUP_PARAMS

Returns

E_ZCL_SUCCESS
E_ZCL_FAIL
E_ZCL_ERR_PARAMETER_NULL
E_ZCL_ERR_INVALID_VALUE

43.7.7 eCLD_CommissioningSetAttribute

```
teZCL_Status eCLD_CommissioningSetAttribute(
        uint8 u8SourceEndPointId,
        teCLD_Commissioning_AttributeSet eAttributeSet,
        void *vptrAttributeSetStructure);
```

Description

This function can be used on a Commissioning cluster server to write values to a particular attribute set of the Commissioning cluster.

Parameters

u8SourceEndPointId Number of the local endpoint through which to issue the request.
eAttributeSet Enumeration indicating attribute set to write to, one of:
E_CLD_COMMISSIONING_ATTR_SET_STARTUP_PARAMS
E_CLD_COMMISSIONING_ATTR_SET_JOIN_PARAMS
E_CLD_COMMISSIONING_ATTR_SET_ENDDEVICE_PARAMS
E_CLD_COMMISSIONING_ATTR_SET_CONCENTRATOR_PARAMS
vptrAttributeSetStructure Pointer to a structure containing the new values for the attribute set - the relevant structures are detailed in Section 43.2

Returns

E_ZCL_SUCCESS
E_ZCL_FAIL
E_ZCL_ERR_PARAMETER_NULL
E_ZCL_ERR_INVALID_VALUE

43.8 Enumerations

43.8.1 teCLD_Commissioning_AttributeID

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

```
typedef enum
{
    E_CLD_COMMISSIONING_ATTR_ID_SHORT_ADDRESS            = 0x0000,
    E_CLD_COMMISSIONING_ATTR_ID_EXT_PANID,
    E_CLD_COMMISSIONING_ATTR_ID_PANID,
    E_CLD_COMMISSIONING_ATTR_ID_CHANNEL_MASK,
    E_CLD_COMMISSIONING_ATTR_ID_PROTOCOL_VERSION,
    E_CLD_COMMISSIONING_ATTR_ID_STACK_PROFILE,
    E_CLD_COMMISSIONING_ATTR_ID_STARTUP_CONTROl,
    E_CLD_COMMISSIONING_ATTR_ID_TC_ADDR                  = 0x0010,
    E_CLD_COMMISSIONING_ATTR_ID_TC_MASTER_KEY,
    E_CLD_COMMISSIONING_ATTR_ID_NETWORK_KEY,
    E_CLD_COMMISSIONING_ATTR_ID_USE_INSECURE_JOIN,
    E_CLD_COMMISSIONING_ATTR_ID_PRECONFIG_LINK_KEY,
    E_CLD_COMMISSIONING_ATTR_ID_NWK_KEY_SEQ_NO,
    E_CLD_COMMISSIONING_ATTR_ID_NWK_KEY_TYPE,
    E_CLD_COMMISSIONING_ATTR_ID_NWK_MANAGER_ADDR,
    E_CLD_COMMISSIONING_ATTR_ID_SCAN_ATTEMPTS            = 0x0020,
    E_CLD_COMMISSIONING_ATTR_ID_TIME_BW_SCANS,
    E_CLD_COMMISSIONING_ATTR_ID_REJOIN_INTERVAL,
    E_CLD_COMMISSIONING_ATTR_ID_MAX_REJOIN_INTERVAL,
    E_CLD_COMMISSIONING_ATTR_ID_INDIRECT_POLL_RATE       = 0x0030,
    E_CLD_COMMISSIONING_ATTR_ID_PARENT_RETRY_THRSHOLD,
    E_CLD_COMMISSIONING_ATTR_ID_CONCENTRATOR_FLAG        = 0x0040,
    E_CLD_COMMISSIONING_ATTR_ID_CONCENTRATOR_RADIUS,
    E_CLD_COMMISSIONING_ATTR_ID_CONCENTRATOR_DISCVRY_TIME,
```

```
} teCLD_Commissioning_AttributeID;;
```

43.8.2 teCLD_Commissioning_AttributeSet

The following structure contains the enumerations used to identify the attribute sets of the Commissioning cluster.

```
typedef enum
{
    E_CLD_COMMISSIONING_ATTR_SET_STARTUP_PARAMS       = 0x00,
    E_CLD_COMMISSIONING_ATTR_SET_JOIN_PARAMS,
    E_CLD_COMMISSIONING_ATTR_SET_ENDDEVICE_PARAMS,
    E_CLD_COMMISSIONING_ATTR_SET_CONCENTRATOR_PARAMS
} teCLD_Commissioning_AttributeSet;
```

43.8.3 teCLD_Commissioning_Command

The following structure contains the enumerations used to identify commands of the Commissioning cluster (the same enumerations are used for requests and their corresponding responses).

```
typedef enum
{
    E_CLD_COMMISSIONING_CMD_RESTART_DEVICE             = 0x00,
    E_CLD_COMMISSIONING_CMD_SAVE_STARTUP_PARAMS,
    E_CLD_COMMISSIONING_CMD_RESTORE_STARTUP_PARAMS,
    E_CLD_COMMISSIONING_CMD_RESET_STARTUP_PARAMS
} teCLD_Commissioning_Command;
```

The above enumerations are described in the table below:

Table 99. Commissioning Command Enumerations

Enumeration	Command
E_CLD_COMMISSIONING_CMD_RESTART_DEVICE	Restart Device request or response
E_CLD_COMMISSIONING_CMD_SAVE_STARTUP_PARAMS	Save Start-up Parameters request or response
E_CLD_COMMISSIONING_CMD_RESTORE_STARTUP_PARAMS	Restore Start-up Parameters request or response
E_CLD_COMMISSIONING_CMD_RESET_STARTUP_PARAMS	Reset Start-up Parameters request or response

43.9 Structures

43.9.1 Attribute Set Structures

The following structures contain the Commissioning cluster attribute sets and are detailed in the referenced sections:

- `tsCLD_StartupParameters` - see Section 43.2.1
- `tsCLD_JoinParameters` - see Section 43.2.2
- `tsCLD_EndDeviceParameters` - see Section 43.2.3
- `tsCLD_ConcentratorParameters` - see Section 43.2.4

43.9.2 tsCLD_Commissioning_RestartDevicePayload

The following structure contains the payload of a Restart Device command.

```
typedef struct
{
    zbmap8                  u8Options;
    uint8                   u8Delay;
    uint8                   u8Jitter;
} tsCLD_Commissioning_RestartDevicePayload;
```

where:

```
u8Options is a 8-bit bitmap specifying the required start-up options:
```

Bits	Option	Description
0-2	Start-up Mode	Determines the starting state of the device restart: • 0b000: Restart with current Start-up Parameter values • 0b001: Restart from existing stack state All other values are reserved.
3	Immediate	Determines how quickly the start-up procedure will begin follow-ing receipt of the command or the specified delay/jitter: • 1: Immediately • 0: At a convenient moment (e.g. following any pending actions)
4-7	-	Reserved

```
u8Delay specifies the time-delay, in seconds, before the start-
up procedure should be executed.
u8Jitter is a value which determines the possible range of values of the jitter that is ad
```

43.9.3 tsCLD_Commissioning_ModifyStartupParametersPayload

The following structure contains the payload of the following commands: Save Start-up Parameters, Restore Start-up Parameters and Reset Start-up Parameters.

```
typedef struct
{
    zbmap8                  u8Options;
    uint8                   u8Index;
} tsCLD_Commissioning_ModifyStartupParametersPayload;
```

where:

```
u8Options is an 8-
bit bitmap specifying the required reset options for the Reset Start-
up Parameters command (it is not used by the other commands):
```

Bits	Option	Description
0	Reset Current	Determines whether the current Start-up Parameters will be reset to their default values: • 1: Reset to default values

Bits	Option	Description
		• 0: Do not reset (remain unchanged)
1	Reset All	Determines whether all stored Start-up Parameter sets will be reset to their default values: • 1: Reset all stored Start-up Parameter sets • 0: Reset the stored Start-up Parameter set with specified index
2	Erase Index	Determines whether the stored Start-up Parameter set with speci-fied index will be erased: • 1: Erase Start-up Parameter set with specified index • 0: Do not erase Start-up Parameter set with specified index
3-7	-	Reserved

```
u8Index is the index of the saved Start-
up Parameter set to which actions specified in u8Options relate (this index is ignored if
```

43.9.4 tsCLD_Commissioning_ResponsePayload

The following structure contains the payload of the responses to the following commands: Save Start-up Parameters, Restore Start-up Parameters and Reset Start-up Parameters.

```
typedef struct
{
    zenum8                  u8Status;
} tsCLD_Commissioning_ResponsePayload;
```

where `u8Status` contains one of the ZCL command status codes listed and described in Section 7.1.4.

43.9.5 tsCLD_CommissioningCustomDataStructure

The Commissioning cluster requires extra storage space to be allocated to be used by internal functions. The structure definition for this storage is shown below:

```
typedef struct
{
    tsZCL_ReceiveEventAddress             sReceiveEventAddress;
    tsZCL_CallBackEvent                   sCustomCallBackEvent;
    tsCLD_CommissioningCallBackMessage    sCallBackMessage;
} tsCLD_CommissioningCustomDataStructure;
```

The fields are for internal use and no knowledge of them is required.

43.9.6 tsCLD_CommissioningCallBackMessage

For a Commissioning 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 `tsCLD_CommissioningCallBackMessage` structure:

```
typedef struct
{
    uint8                       u8CommandId;
    union
```

```
    {
        tsCLD_Commissioning_RestartDevicePayload
                            *psRestartDevicePayload;
        tsCLD_Commissioning_ModifyStartupParametersPayload
                            *psModifyStartupParamsPayload;
    } uReqMessage;
    union
    {
        tsCLD_Commissioning_ResponsePayload
                            *psCommissioningResponsePayload;
    } uRespMessage;
} tsCLD_CommissioningCallBackMessage;
```

where:

- `u8CommandId` indicates the type of Commissioning command that has been received by a cluster server or client (the same enumerations are used for requests on the server and responses on the client), one of:
  - E_CLD_COMMISSIONING_CMD_RESTART_DEVICE
  - E_CLD_COMMISSIONING_CMD_SAVE_STARTUP_PARAMS
  - E_CLD_COMMISSIONING_CMD_RESTORE_STARTUP_PARAMS
  - E_CLD_COMMISSIONING_CMD_RESET_STARTUP_PARAMS
- `uReqMessage` is a union containing the payload of a request command in the following form:
  - `psRestartDevicePayload` is a pointer to a structure containing the Restart Device command payload - see Section 43.9.2
  - `psModifyStartupParamsPayload` is a pointer to a structure containing the (common) payload for the Save Start-up Parameters, Restore Start-up Parameters and Reset Start-up Parameters commands - see Section 43.9.3
- `uRespMessage` is a union containing the payload of a response command in the following form:
  - `psCommissioningResponsePayload` is a pointer to a structure containing the (common) payload for the Save Start-up Parameters, Restore Start-up Parameters and Reset Start-up Parameters responses - see Section 43.9.4

For further information on Commissioning cluster events, refer to Section 43.6.

43.10 Compile-time options

To enable the Commissioning cluster in the code to be built, it is necessary to add the following to the **zcl_options.h** file:

```
#define CLD_COMMISSIONING
```

In addition, to include the software for a cluster client or server or both, it is necessary to add one or both of the following to the same file:

```
#define COMMISSIONING_CLIENT
#define COMMISSIONING_SERVER
```

Optional Attributes

The Commissioning cluster contains attributes that may be optionally enabled at compile-time by adding some or all of the following lines to the **zcl_options.h** file (see Section 43.2 and Section 43.3):

```
#define         CLD_COMM_ATTR_SHORT_ADDRESS
#define CLD_COMM_ATTR_EXTENED_PAN_ID
```

```
#define CLD_COMM_ATTR_PAN_ID
#define CLD_COMM_ATTR_CHANNEL_MASK
#define CLD_COMM_ATTR_PROTOCOL_VERSION
#define CLD_COMM_ATTR_STACK_PROFILE
#define CLD_COMM_ATTR_START_UP_CONTROL
        #define CLD_COMM_ATTR_TC_ADDR
#define CLD_COMM_ATTR_TC_MASTER_KEY
        #define CLD_COMM_ATTR_NWK_KEY
#define CLD_COMM_ATTR_USE_INSECURE_JOIN
        #define CLD_COMM_ATTR_PRE_CONFIG_LINK_KEY
#define CLD_COMM_ATTR_NWK_KEY_SEQ_NO
        #define CLD_COMM_ATTR_NWK_KEY_TYPE
#define CLD_COMM_ATTR_NWK_MANAGER_ADDR
#define CLD_COMM_ATTR_SCAN_ATTEMPTS
#define CLD_COMM_ATTR_TIME_BW_SCANS
#define CLD_COMM_ATTR_REJOIN_INTERVAL
#define CLD_COMM_ATTR_MAX_REJOIN_INTERVAL
        #define CLD_COMM_ATTR_INDIRECT_POLL_RATE
#define CLD_COMM_ATTR_PARENT_RETRY_THRSHLD
#define CLD_COMM_ATTR_CONCENTRATOR_FLAG
#define CLD_COMM_ATTR_CONCENTRATOR_RADIUS
#define CLD_COMM_ATTR_CONCENTRATOR_DISCVRY_TIME
```

Optional Commands

The Commissioning cluster contains commands that may be optionally enabled at compile-time by adding some or all of the following lines to the **zcl_options.h** file.

To enable the Save Start-up Parameters command, add the following line:

```
#define CLD_COMMISSIONING_CMD_SAVE_STARTUP_PARAMS
```

To enable the Restore Start-up Parameters command, add the following line:

```
#define CLD_COMMISSIONING_CMD_RESTORE_STARTUP_PARAMS
```

44 Touchlink Commissioning Cluster

This chapter describes the Touchlink Commissioning cluster, which can be used when forming a network or adding a new node to an existing network.

The Touchlink Commissioning cluster has a Cluster ID of 0x1000.

44.1 Overview

The Touchlink Commissioning cluster is associated with a node as a whole, rather than with individual ZigBee devices on the node. It must be used on nodes that incorporate one or more of the ZigBee devices indicated in Table 83 below, which shows the supported devices when the Touchlink Commissioning cluster acts as a client, server and combined client/server.

Table 100. Touchlink Commissioning Cluster in ZigBee Devices

Client	Client/Server	Server
Colour Controller	Colour Controller	Colour Controller
Colour Scene Controller	Colour Scene Controller	Colour Scene Controller
Non-Colour Controller	Non-Colour Controller	Non-Colour Controller
Non-Colour Scene Controller	Non-Colour Scene Controller	Non-Colour Scene Controller
Control Bridge	Control Bridge	Control Bridge
On/Off Sensor	On/Off Sensor	On/Off Sensor
	On/Off Light	On/Off Light
	On/Off Plug-in Unit	On/Off Plug-in Unit
	Dimmable Light	Dimmable Light
	Dimmable Plug-in Unit	Dimmable Plug-in Unit
	Colour Light	Colour Light
	Extended Colour Light	Extended Colour Light
	Colour Temperature Light	Colour Temperature Light

This cluster supports two sets of functionality, corresponding to two distinct commands sets:

- Touchlink
- Commissioning Utility

Functions are provided for implementing both sets of commands. These functions are referenced in Section 44.4 and Section 44.5, and detailed in Section 44.7.

The Commissioning Utility functionality is not required on Lighting devices.

For the compile-time options for enabling the Touchlink Commissioning cluster for Touchlink and the Commissioning Utility, refer to Section 44.10.

44.2 Cluster structure and attributes

This cluster has no attributes, as a server or a client. Therefore, the cluster structure `tsCLD_ZllCommission` is referred to using a null pointer.

44.3 Commissioning operations

Commissioning involves forming a network or adding a new node to an existing network. A node from which commissioning can be initiated is referred to as an ‘initiator’ - this may be a remote control unit, but could also be a lamp.

- An ‘initiator’ node must support the Touchlink Commissioning cluster as a client.
- A node to be added to the network must support the Touchlink Commissioning cluster as a server (or as both a server and client).

Note that commissioning a new network involves adding at least one node to the new network (as well as the initiator).

Commissioning may involve two stages, depending on the type of node added to the network by the initiator:

1. The node is added to the network using the Touchlink commands of the Touchlink Commissioning cluster. In practice for the user, this typically involves bringing the initiator node physically close to the target node and pressing a button.

2. If the initiator node and the new node will both be used to control lights in the network, the new node must learn certain information (such as controlled endpoints and configured groups) from the initiator. This exchange of information uses the Commissioning Utility commands of the Touchlink Commissioning cluster.

Note: ***Note:*** *The Touchlink Commissioning cluster instance for Touchlink must reside on its own endpoint on a node. Therefore, a Touchlink commissioning application must be provided which is distinct from the main application. However, the cluster instance for the Commissioning Utility can reside on the same endpoint as the main application (and be used in this application).*

Commissioning using the supplied functions for Touchlink and the Commissioning Utilty is described in Section 44.4 and Section 44.5.

44.4 Using Touchlink

Touchlink is used for the basic commissioning of a new network or adding a new node to an existing network. A dedicated Touchlink application (which is distinct from the main application on the node) must reside on its own endpoint. This requires:

- a Touchlink Commissioning cluster instance as a client to be created on the endpoint on the initiator node.
- a Touchlink Commissioning cluster instance as a server to be created on the endpoint on the target node.

The initiator node also requires a Touchlink Commissioning cluster instance as a server (on the same endpoint), since the node also needs the capability to join an existing network.

An endpoint is registered for Touchlink (on both nodes) using the function **eZLL_RegisterCommissionEndPoint()**. This function also creates a Touchlink Commissioning cluster instance of the type (server, client or both) determined by the compile-time options in the header file **zcl_options.h** (see Section 44.10).

The initiator must then send a sequence of request commands to the target node. The Touchlink request command set is summarized in Table 101. Touchlink functions for issuing these commands are provided and are detailed in Section 44.7.1.

Table 101. Touchlink Request Commands

Command	Identifier	Description
Scan Request *	0x00	Requests other devices (potential nodes) in the local neighbourhood to respond. A scan request is first performed on channel 11, up to five times until a response is received. If no response is received, a scan request is then performed once on each of channels 15, 20 and 25, and then the remaining channels (12, 13, 14, 16, etc) until a response is detected.
Device Information Request *	0x02	Requests information about the devices on a remote node
Identify Request	0x06	Requests a remote node to physically identify itself (for example, visually by flashing an LED)

Table 101. Touchlink Request Commands*...continued*

Command	Identifier	Description
Reset To Factory New Request	0x07	Requests a factory reset of a remote node
Network Start Request *	0x10	Requests a new network to be created comprising the initiator and a detected Router
Network Join Router Request *	0x12	Requests a Router to join the network
Network Join End Device Request *	0x14	Requests an End Device to join the network
Network Update Request *	0x16	Requests an update of the network settings on a remote node (if the supplied Network Update Identifier is more recent than the one on the node)

* These commands have corresponding responses.

All Touchlink commands are sent as inter-PAN messages.

Use of the above commands and associated functions is described in the sub-sections below.

44.4.1 Creating a network

A network is formed from an initiator node and a Router node (usually the initiator is an End Device and will have no routing capability in the network). The Touchlink network creation process is described below and is illustrated in Figure 7 (also refer to the command list in Table 101).

Note: *Received Touchlink requests and responses are handled as ZigBee PRO events. The event handling is not detailed below but is outlined in Section 44.6.*

1. **Scan Request:** The initiator sends a Scan Request to nodes in its vicinity. The required function is:

   eCLD_ZllCommissionCommandScanReqCommandSend()

2. **Scan Response:** A receiving node replies to the Scan Request by sending a Scan Response, which includes the device type of the responding node (e.g. Router). The required function is:

   eCLD_ZllCommissionCommandScanRspCommandSend()

3. **Device Information Request:** The initiator sends a Device Information Request to the detected Routers that are of interest. The required function is:

   eCLD_ZllCommissionCommandDeviceInfoReqCommandSend()

4. **Device Information Response:** A receiving Router replies to the Device Information Request by sending a Device Information Response. The required function is:

   eCLD_ZllCommissionCommandDeviceInfoRspCommandSend()

5. **Identify Request (Optional):** The initiator may send an Identify Request to the node which has been chosen as the first Router of the new network, in order to confirm that the correct physical node is being commissioned. The required function is:

   eCLD_ZllCommissionCommandDeviceIdentifyReqCommandSend()

6. **Network Start Request:** The intiator sends a Network Start Request to the chosen Router in order to create and start the network. The required function is:

   eCLD_ZllCommissionCommandNetworkStartReqCommandSend()

7. **Network Start Response:** The Router replies to the Network Start Request by sending a Network Start Response. The required function is:

   eCLD_ZllCommissionCommandNetworkStartRspCommandSend()

Once the Router has started the network, the initiator joins the network (Router). The initiator then collects endpoint and cluster information from the Lighting device(s) on the Router node, and stores this information in a local lighting database.

Once the network (consisting of the initiator and one Router) is up and running, further nodes may be added as described in Section 44.4.2.

Table 102. Creating a Network

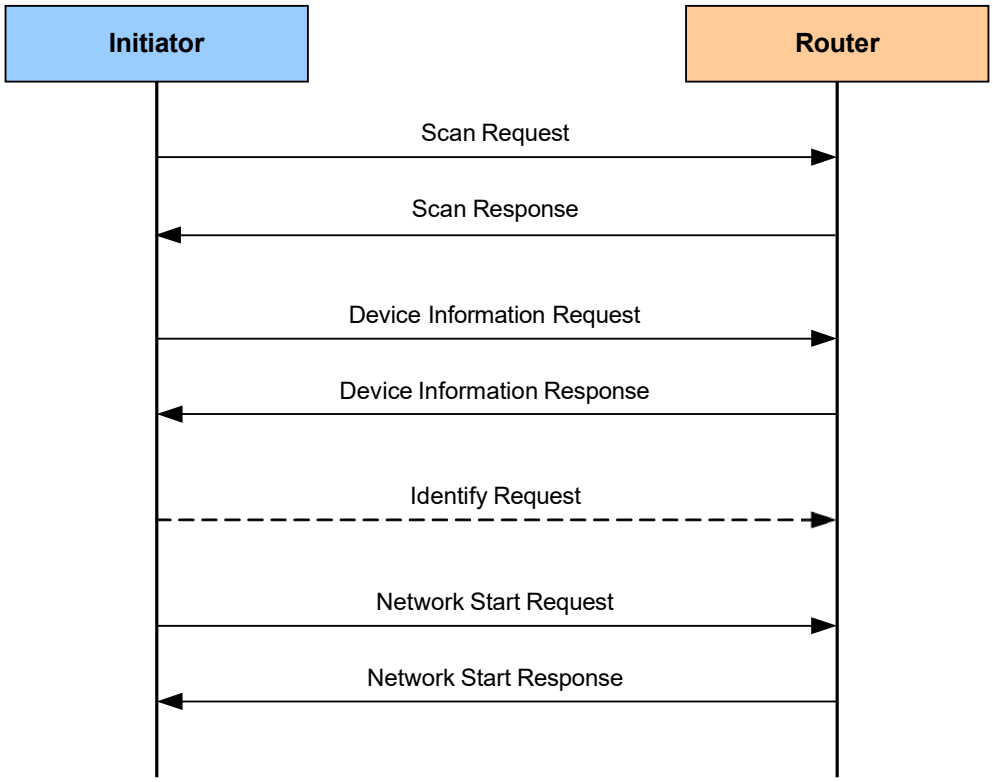


Figure 7. Creating a Network

44.4.2 Adding to an existing network

A network (that has been set up as described in Section 44.4.1) can be extended by adding a node. The Touchlink extension process is described below and illustrated in Figure 8 (also refer to the command list in Table 101).

Note: *Received Touchlink requests and responses are handled as ZigBee PRO events. The event handling is not detailed below but is outlined in Section 44.6.*

1. **Scan Request:** The initiator sends a Scan Request to nodes in its vicinity. The required function is:

   eCLD_ZllCommissionCommandScanReqCommandSend()

2. **Scan Response:** A receiving node replies to the Scan Request by sending a Scan Response. The required function is:

   eCLD_ZllCommissionCommandScanRspCommandSend()

3. **Device Information Request:** The initiator sends a Device Information Request to those detected nodes that are of interest. The required function is:

   eCLD_ZllCommissionCommandDeviceInfoReqCommandSend()

4. **Device Information Response:** A receiving node replies to the Device Information Request by sending a Device Information Response. The required function is:

eCLD_ZllCommissionCommandDeviceInfoRspCommandSend()

5. **Identify Request (Optional):** The initiator may send an Identify Request to the node which has been chosen to be added to the network, in order to confirm that the correct physical node is being commissioned. The required function is:

eCLD_ZllCommissionCommandDeviceIdentifyReqCommandSend()

6. **Network Join Request:** Depending on the target node type, the initiator sends a Network Join Router Request or Network Join End Device Request, as appropriate, to the target node. The required function is one of:

eCLD_ZllCommissionCommandNetworkJoinRouterReqCommandSend()
eCLD_ZllCommissionCommandNetworkJoinEndDeviceReqCommandSend()

7. **Network Join Response:** Depending on the receiving node type, the node replies to the join request by sending a Network Join Router Response or Network Join End Device Response. The required function is one of:

eCLD_ZllCommissionCommandNetworkJoinRouterRspCommandSend()
eCLD_ZllCommissionCommandNetworkJoinEndDeviceRspCommandSend()

The node should now be a member of the network. The initiator then collects endpoint and cluster information from any Lighting device(s) on the new node, and stores this information in its local lighting database.

If the new node is to be used to control the light nodes of the network then it will need to learn certain information (such as controlled endpoints and configured groups) from the initiator - this is done using the Commissioning Utility commands, as described in Section 44.5.

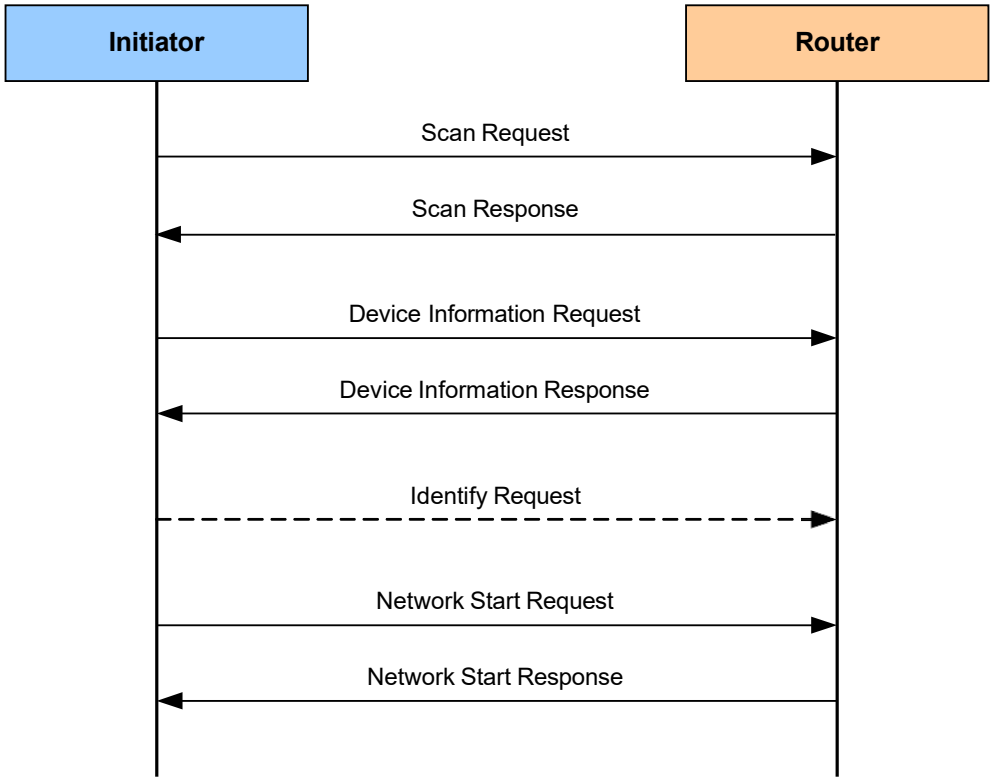


Figure 8. Extending a Network (Adding a Node)

44.4.3 Updating network settings

If one or more of the network settings change (e.g. the radio channel used), all nodes of the network need to be updated with the new settings.

To allow nodes to keep track of the status of the network settings, the Network Update Identifier is used. This identifier takes a value in the range 0x00 to 0xFF and is incremented when a network update has occurred (the value wraps around at 0xFF).

A node can be instructed to update its network settings by sending a Network Update Request to it. The required function is:

eCLD_ZllCommissionCommandNetworkUpdateReqCommandSend()

The payload of the sent command contains the latest network settings and the current value of the Network Update Identifier (see Section 44.8.17). If the payload value is more recent than the value held by the target node, the node should update its network settings with those in the payload.

44.4.4 Stealing a node

A node that is already part of a network can be taken or ‘stolen’ by another network using Touchlink (in which case, the stolen node will cease to be a member of its previous network). This transfer can only be performed on a node which supports one or more Lighting devices (and not Controller devices).

The node is stolen using an initiator in the new network, e.g. from a remote control unit. The ‘stealing’ process is as follows:

1. The initiator sends a Scan Request to nodes in its vicinity. The required function is:

eCLD_ZllCommissionCommandScanReqCommandSend()

2. A receiving node replies to the Scan Request by sending a Scan Response. The required function is:

eCLD_ZllCommissionCommandScanRspCommandSend()

3. The initiator receives Scan Responses from one or more nodes and, based on these responses, selects a node (containing a Lighting device) that is already a member of another network.

4. The initiator then sends a Reset To Factory New Request to the desired node. The required function is:

eCLD_ZllCommissionCommandFactoryResetReqCommandSend()

5. On receiving this request on the target node, the event E_CLD_COMMISSION_CMD_FACTORY_RESET_REQ is generated and the function **ZPS_eAplZdoLeaveNetwork()** should be called. In addition, all persistent data should be reset.

6. The node can then be commissioned into the new network by following the process in Section 44.4.2 from Step3.

Alternatively, instead of following the above process, a node can be stolen by either:

- Following the full process for creating a network in Section 44.4.1 and calling **ZPS_eAplZdoLeaveNetwork()** on the target node when a Network Start Request is received.
- Following the full process for adding a node in Section 44.4.2 and calling **ZPS_eAplZdoLeaveNetwork()** on the target node when a Network Join Router Request or Network Join End Device Request is received.

Note: *If a node containing a Controller device (e.g. a remote control unit) is to be used in another network, it must first be reset using a Reset To Factory New Request. It can then be used to create a new network (see Section 44.4.1) or to learn the control information of an existing network (see Section 44.5).*

44.5 Using the Commissioning Utility

The Commissioning Utility is used when a network node needs to learn lighting control information (such as controlled endpoints and configured groups) from another node in the network. It is typically used when a new remote control unit is introduced into the network and needs to learn information from an existing remote control unit.

Unlike Touchlink, the Commissioning Utility can be incorporated in the main application on the node (and therefore use the same endpoint). This requires:

- a Touchlink Commissioning cluster instance as a client to be created on the endpoint on the ‘learner’ node
- a Touchlink Commissioning cluster instance as a server to be created on the endpoint on the ‘teacher’ node

A Touchlink Commissioning cluster instance for the Commissioning Utility can be created using the function **eCLD_ZllUtilityCreateUtility()**, on both nodes.

It is the responsibility of the learner node to request the required information from the teacher node. The Commissioning Utility command set is summarised in Table 85. Commissioning Utility functions for issuing these commands are provided and are detailed in Section 44.7.2.

Table 103. Commissioning Utility commands

Command	Identifier	Description
Endpoint information	0x40	Sends information about local endpoint (from teacher to learner)
Get Group Identifiers Request	0x41	Requests Group information from a remote node (from learner to teacher)
Get Endpoint List Request	0x42	Requests endpoint information from a remote node

Table 103. Commissioning Utility commands*...continued*

Command	Identifier	Description
		(from learner to teacher)

Use of the above commands and associated functions is described below and is illustrated in Figure 9.

Note: *Received Commissioning Utility requests and responses are handled as ZigBee PRO events by the ZCL (this event handling is therefore transparent to the application).*

1. **Endpoint Information command:** The teacher node first sends an Endpoint Information command containing basic information about its local endpoint (IEEE address, network address endpoint number, Profile ID, Device ID) to the learner node. The required function is:

   eCLD_ZllUtilityCommandEndpointInformationCommandSend()
   Note that the teacher node will already have the relevant target endpoint on the learner node from the joining process (described in Section 44.4).

2. **Get Endpoint List Request:** The learner node then sends a Get Endpoint List Request to the teacher node to request information about the remote endpoints that the teacher node controls. The required function is:

   eCLD_ZllUtilityCommandGetEndpointListReqCommandSend()
   The teacher node automatically replies to the Get Endpoint List Request by sending a Get Endpoint List Response containing the requested information.

3. **Get Group Identifiers Request:** The learner node then sends a Get Group Identifiers Request to the teacher node to request a list of the lighting groups configured on the teacher node. The required function is:

   eCLD_ZllUtilityCommandGetGroupIdReqCommandSend()
   The teacher node automatically replies to the Get Group Identifiers Request by sending a Get Group Identifiers Response containing the requested information.

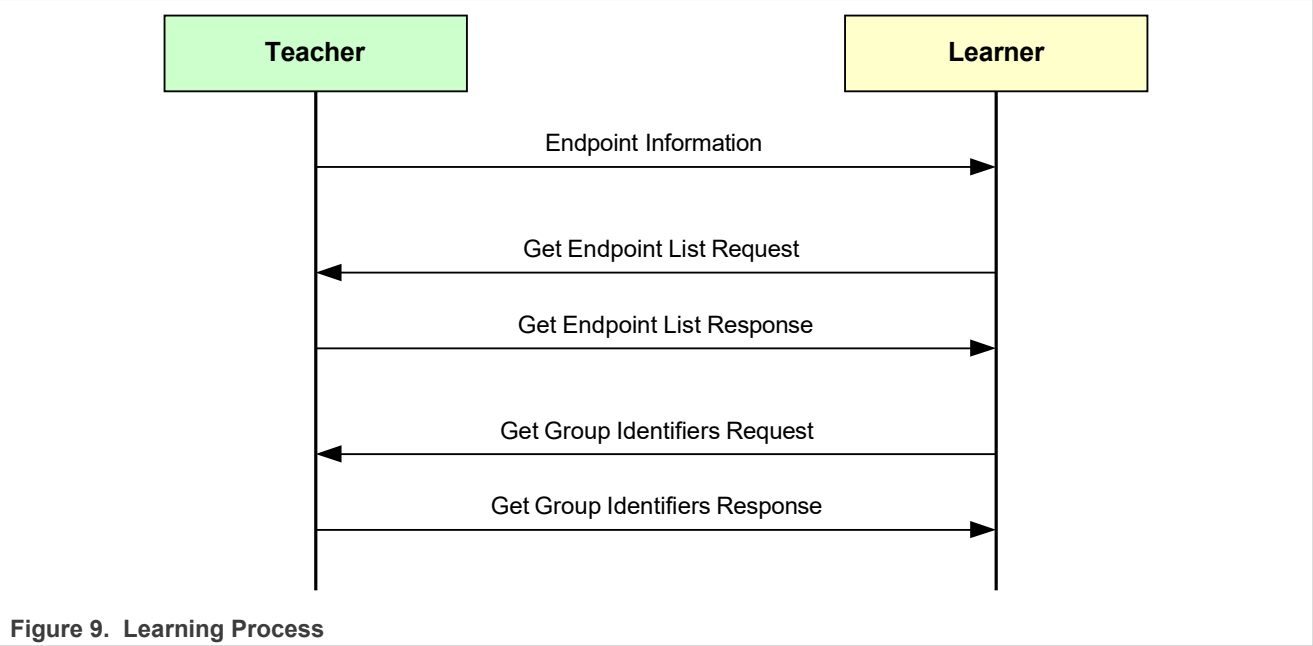


Figure 9. Learning Process

To complete the learning process, the learner node may need other information which can be acquired using commands/functions of the relevant cluster.

44.6 Touchlink Commissioning events

Touchlink Commissioning cluster events that result from receiving Touchlink requests and responses must be handled at the application level (while events that result from Commissioning Utility requests and responses are handled by the ZCL).

When a Touchlink request or response command (e.g. a Scan Request) is received by a node, a stack event is generated which is wrapped in a `tsZCL_CallBackEvent` structure. In this structure:

- `eEventType` field is set to E_ZCL_CBET_CLUSTER_CUSTOM
- `sClusterCustomMessage` field's `tsZCL_ClusterCustomMessage` structure is filled in by:
  - setting `u16ClusterId` to ZLL_CLUSTER_ID_COMMISSIONING
  - pointing `pvCustomData` to the payload data of the received command

    The above structure is described in Section 6.1.15.

The payload data contains a command ID, which uses one of the enumerations listed in Section 44.6.1. The event is passed to the ZCL event handler which checks that the command ID is valid for the target endpoint. If it is valid, the user-defined callback function is invoked that was specified through the function **eZLL_RegisterCommissionEndPoint()**. The callback function can access the payload through the `tsCLD_ZllCommissionCustomDataStructure` structure, which is created when the above function is called.

Thus, the above user-defined callback function must be designed to handle the relevant Touchlink events:

- For a request, the callback function may need to populate a structure with the required data and send a response using the appropriate response function, e.g. by calling **eCLD_ZllCommissionCommandScanRspCommandSend()** to respond to a Scan Request.
- For a response, the callback function may just need to extract the returned data from the event.

Alternatively, the callback function may simply notify the main application of the received command and provide the payload, so that the application can process the command.

44.6.1 Touchlink command events

The events that can be generated for Touchlink are listed and described below (the enumerations are defined in the structure `teCLD_ZllCommission_Command`, shown in Section 44.9.1).

Table 104. Touchlink Events

Event	Description
E_CLD_COMMISSION_CMD_SCAN_REQ	A Scan Request has been received (by server)
E_CLD_COMMISSION_CMD_SCAN_RSP	A Scan Response has been received (by client)
E_CLD_COMMISSION_CMD_DEVICE_INFO_REQ	A Device Information Request has been received (by server)
E_CLD_COMMISSION_CMD_DEVICE_INFO_RSP	A Device Information Response has been received (by client)
E_CLD_COMMISSION_CMD_IDENTIFY_REQ	An Identify Request has been received (by server)
E_CLD_COMMISSION_CMD_FACTORY_RESET_REQ	A Reset To Factory New Request has been received (by server)
E_CLD_COMMISSION_CMD_NETWORK_START_REQ	A Network Start Request has been received (by server)
E_CLD_COMMISSION_CMD_NETWORK_START_RSP	A Network Start Response has been received (by cli-ent)
E_CLD_COMMISSION_CMD_NETWORK_JOIN_ROUTER_REQ	A Network Join Router Request has been received (by server)

Table 104. Touchlink Events...*continued*

Event	Description
E_CLD_COMMISSION_CMD_NETWORK_JOIN_ROUTER_RSP	A Network Join Router Response has been received (by client)
E_CLD_COMMISSION_CMD_NETWORK_JOIN_END_DEVICE_REQ	A Network Join End Device Request has been received (by server)
E_CLD_COMMISSION_CMD_NETWORK_JOIN_END_DEVICE_RSP	A Network Join End Device Response has been received (by client)
E_CLD_COMMISSION_CMD_NETWORK_UPDATE_REQ	A Network Update Request has been received (by server)

44.6.2 Commissioning Utility Command Events

The events that can be generated for the Commissioning Utility are listed and described below (the enumerations are defined in the structure `teCLD_ZllUtility_Command`, shown in Section 44.9.2).

Table 105. Touchlink Events

Event	Description
E_CLD_UTILITY_CMD_ENDPOINT_INFO	An Endpoint Information command has been received (by client)
E_CLD_UTILITY_CMD_GET_GROUP_ID_REQ_RSP	A Get Group Identifiers Request has been received (by server) or a Get Group Identifiers Response has been received (by client)
E_CLD_UTILITY_CMD_GET_ENDPOINT_LIST_REQ_RSP	A Get Endpoint List Request has been received (by server) or a Get Endpoint List Response has been received (by client)

44.7 Functions

The functions of the Touchlink Commissioning cluster are divided into two categories:

- Touchlink functions, detailed in Section 44.7.1
- Commissioning Utility functions, detailed in Section 44.7.2

44.7.1 Touchlink functions

The following Touchlink functions are provided:

1. eZLL_RegisterCommissionEndPoint
2. eCLD_ZllCommissionCreateCommission
3. eCLD_ZllCommissionCommandScanReqCommandSend
4. eCLD_ZllCommissionCommandScanRspCommandSend
5. eCLD_ZllCommissionCommandDeviceInfoReqCommandSend
6. eCLD_ZllCommissionCommandDeviceInfoRspCommandSend
7. eCLD_ZllCommissionCommandDeviceIdentifyReqCommandSend
8. eCLD_ZllCommissionCommandFactoryResetReqCommandSend
9. eCLD_ZllCommissionCommandNetworkStartReqCommandSend
10. eCLD_ZllCommissionCommandNetworkStartRspCommandSend
11. eCLD_ZllCommissionCommandNetworkJoinRouterReqCommandSend
12. eCLD_ZllCommissionCommandNetworkJoinRouterRspCommandSend

13. eCLD_ZllCommissionCommandNetworkJoinEndDeviceReqCommandSend
14. eCLD_ZllCommissionCommandNetworkJoinEndDeviceRspCommandSend
15. eCLD_ZllCommissionCommandNetworkUpdateReqCommandSend

44.7.1.1 eZLL_RegisterCommissionEndPoint

```
teZCL_Status eZLL_RegisterCommissionEndPoint(
        uint8 u8EndPointIdentifier,
        tfpZCL_ZCLCallBackFunction cbCallBack,
        tsZLL_CommissionEndpoint *psDeviceInfo);
```

Description

This function registers a ‘commissioning’ endpoint for Touchlink and creates a Touchlink Commissioning cluster instance on the endpoint.

Touchlink must have its own application (separate from the main application) on its own endpoint.

This function uses **eCLD_ZllCommissionCreateCommission()** to create the cluster instance. The type of cluster instance to be created (server or client, or both) is determined using the compile-time options in the header file **zcl_options.h** (refer to Section 44.10).

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 Touchlink endpoint information (see Section 44.8.1)

Returns

E_ZCL_SUCCESS

44.7.1.2 eCLD_ZllCommissionCreateCommission

```
teZCL_Status eCLD_ZllCommissionCreateCommission(
        tsZCL_ClusterInstance *psClusterInstance,
        bool_t bIsServer,
        tsZCL_ClusterDefinition *psClusterDefinition,
        void *pvSharedStructPtr,
        tsZCL_AttributeStatus *psAttributeStatus,
        tsCLD_ZllCommissionCustomDataStructure
          *psCustomDataStructure);
```

Description

This function creates a Touchlink Commissioning cluster instance for Touchlink on the endpoint of the calling application. The type of cluster instance (server or client) to be created must be specified.

In practice, this function does not need to be called explicitly by the application, as the function **eZLL_RegisterCommissionEndPoint()** calls this function to create the cluster instance.

Parameters

psClusterInstance Pointer to cluster instance structure on local endpoint
bIsServer Type of cluster instance (server or client) to be created:
TRUE - server
FALSE - client
psClusterDefinition Pointer to cluster definition structure containing information about the cluster
pvSharedStructPtr Pointer to structure containing the shared storage for the cluster
psAttributeStatus Pointer to a structure containing the storage for each attribute's status
psCustomDataStructure Pointer to custom data to be provided to the cluster (see Section 44.8.3)

Returns

E_ZCL_SUCCESS

44.7.1.3 eCLD_ZllCommissionCommandScanReqCommandSend

```
teZCL_Status                eCLD_ZllCommissionCommandScanReqCommandSend(
        ZPS_tsInterPanAddress *psDestinationAddress,
        uint8 *pu8TransactionSequenceNumber,
        tsCLD_ZllCommission_ScanReqCommandPayload
 *psPayload);
```

Description

This function is used to send a Scan Request command to initiate a scan for other nodes in the local neighbourhood. The command is sent as an inter-PAN message.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is 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

psDestinationAddress Pointer to stucture containing PAN ID and address information for target node(s)
pu8TransactionSequenceNumber Pointer to a location to store the Transaction Sequence Number (TSN) of the request
psPayload Pointer to structure containing payload data for the Scan Request command (see Section 44.8.5)

Returns

E_ZCL_SUCCESS

44.7.1.4 eCLD_ZllCommissionCommandScanRspCommandSend

```
PUBLIC teZCL_Status eCLD_ZllCommissionCommandScanRspCommandSend(
        ZPS_tsInterPanAddress *psDestinationAddress,
        uint8 *pu8TransactionSequenceNumber,
        tsCLD_ZllCommission_ScanRspCommandPayload
```

```
 *psPayload);
```

Description

This function is used to send a Scan Response command containing information about the local node in reply to a received Scan Request from a remote node. The command is sent as an inter-PAN message.

A pointer must be provided to a structure containing the data to be returned.

The specified Transaction Sequence Number (TSN) of the response must match the TSN of the corresponding request, as this will allow the response to be paired with the request at the destination.

Parameters

psDestinationAddress Pointer to stucture containing PAN ID and address information for target node

pu8TransactionSequenceNumber Pointer to location containing the Transaction Sequence Number (TSN) of the response

psPayload Pointer to structure containing payload data for the Scan Response command (see Section 44.8.6)

Returns

E_ZCL_SUCCESS

44.7.1.5 eCLD_ZllCommissionCommandDeviceInfoReqCommandSend

```
teZCL_Status eCLD_ZllCommissionCommandDeviceInfoReqCommandSend(
ZPS_tsInterPanAddress *psDestinationAddress,
uint8 *pu8TransactionSequenceNumber, tsCLD_ZllCommission_DeviceInfoReqCommandPayload
 *psPayload);
```

Description

This function is used to send a Device Information Request command to obtain information about the devices on a remote node. The command is sent as an inter-PAN message.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is 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

psDestinationAddress Pointer to stucture containing PAN ID and address information for target node

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

psPayload Pointer to structure containing payload data for the Device Information Request command (see Section 44.8.7)

Returns

E_ZCL_SUCCESS

44.7.1.6 eCLD_ZllCommissionCommandDeviceInfoRspCommandSend

```
PUBLIC teZCL_Status eCLD_ZllCommissionCommandDeviceInfoRspCommandSend(
        ZPS_tsInterPanAddress *psDestinationAddress,
        uint8 *pu8TransactionSequenceNumber,
        tsCLD_ZllCommission_DeviceInfoRspCommandPayload
 *psPayload);
```

Description

This function is used to send a Device Information Response command containing information about the devices on the local node in reply to a received Device Information Request from a remote node. The command is sent as an inter-PAN message.

A pointer must be provided to a structure containing the data to be returned.

The specified Transaction Sequence Number (TSN) of the response must match the TSN of the corresponding request, as this will allow the response to be paired with the request at the destination.

Parameters

psDestinationAddress Pointer to stucture containing PAN ID and address information for target node

pu8TransactionSequenceNumber Pointer to location containing the Transaction Sequence Number (TSN) of the response

psPayload Pointer to structure containing payload data for the Device Information Response command (see Section 44.8.8)

Returns

E_ZCL_SUCCESS

44.7.1.7 eCLD_ZllCommissionCommandDeviceIdentifyReqCommandSend

abcdef

```
teZCL_Status eCLD_ZllCommissionCommandDeviceIdentifyReqCommandSend(
  ZPS_tsInterPanAddress *psDestinationAddress,
  uint8 *pu8TransactionSequenceNumber,
  tsCLD_ZllCommission_IdentifyReqCommandPayload
 *psPayload);
```

Description

This function is used to send an Identify Request command to ask a remote node to identify itself by entering ‘identify mode’ (this is a visual indication, such as flashing a LED). The command is sent as an inter-PAN message.

The command payload contains a value indicating the length of time, in seconds, that the target device should remain in identify mode. It is also possible to use this command to instruct the target node to immediately exit identify mode (if it is already in this mode).

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is 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

psDestinationAddress Pointer to stucture containing PAN ID and address information for target node

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

psPayload Pointer to structure containing payload data for the Identify Request command (see Section 44.8.9)

Returns

E_ZCL_SUCCESS

44.7.1.8 eCLD_ZllCommissionCommandFactoryResetReqCommandSend

```
teZCL_Status eCLD_ZllCommissionCommandFactoryResetReqCommandSend(
        ZPS_tsInterPanAddress *psDestinationAddress,
        uint8 *pu8TransactionSequenceNumber,
        tsCLD_ZllCommission_FactoryResetReqCommandPayload
 *psPayload);
```

Description

This function is used to send a Reset to Factory New Request command to ask a remote node to return to its ‘factory new’ state. The command is sent as an inter-PAN message.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is 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

psDestinationAddress Pointer to stucture containing PAN ID and address information for target node

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

psPayload Pointer to structure containing payload data for the Reset to Factory New Request command (see Section 44.8.10)

Returns

E_ZCL_SUCCESS

44.7.1.9 eCLD_ZllCommissionCommandNetworkStartReqCommandSend

```
teZCL_Status eCLD_ZllCommissionCommandNetworkStartReqCommandSend(
        ZPS_tsInterPanAddress *psDestinationAddress,
        uint8 *pu8TransactionSequenceNumber,
        tsCLD_ZllCommission_NetworkStartReqCommandPayload
```

```
 *psPayload);
```

Description

This function is used to send a Network Start Request command to create a new network with a detected Router. The command is sent as an inter-PAN message.

The function is called once the results of a Scan Request command have been received and a detected Router has been selected.

The command payload contains information about the network and the local node, as well as certain data for the target node. This payload information is detailed in Section 44.8.11.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is 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

psDestinationAddress Pointer to stucture containing PAN ID and address information for target node

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

psPayload Pointer to structure containing payload data for the Network Start Request command (see Section 44.8.11)

Returns

E_ZCL_SUCCESS

44.7.1.10 eCLD_ZllCommissionCommandNetworkStartRspCommandSend

```
PUBLIC teZCL_Status         eCLD_ZllCommissionCommandNetworkStartRspCommandSend(
        ZPS_tsInterPanAddress *psDestinationAddress,
        uint8 *pu8TransactionSequenceNumber,
        tsCLD_ZllCommission_NetworkStartRspCommandPayload
 *psPayload);
```

Description

This function is used to send a Network Start Response command to confirm that the local (Router) node is ready to be the first node to join a newly created network in reply to a received Network Start Request from a remote node. The command is sent as an inter-PAN message.

A pointer must be provided to a structure containing the data to be returned.

The specified Transaction Sequence Number (TSN) of the response must match the TSN of the corresponding request, as this will allow the response to be paired with the request at the destination.

Parameters

psDestinationAddress Pointer to stucture containing PAN ID and address information for target node

pu8TransactionSequenceNumber Pointer to location containing the Transaction Sequence Number (TSN) of the response

psPayload Pointer to structure containing payload data for the Network Start Response command (see Section 44.8.12)

Returns

E_ZCL_SUCCESS

44.7.1.11 eCLD_ZllCommissionCommandNetworkJoinRouterReqCommandSend

```
teZCL_Status eCLD_ZllCommissionCommandNetworkJoinRouterReqCommandSend(
        ZPS_tsInterPanAddress *psDestinationAddress,
        uint8 *pu8TransactionSequenceNumber,
        tsCLD_ZllCommission_NetworkJoinRouterReqCommandPayload
 *psPayload);
```

Description

This function is used to send a Network Join Router Request command to allow a detected Router to join the created network. The command is sent as an inter-PAN message.

The function can be called once a network has been created. The target Router is distinct from the Router that was included when network was created.

The command payload contains information about the network and the local node, as well as certain data for the target node. This payload information is detailed in Section 44.8.13.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is 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

psDestinationAddress Pointer to stucture containing PAN ID and address information for target node

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

psPayload Pointer to structure containing payload data for the Network Join Router Request command (see Section 44.8.13)

Returns

E_ZCL_SUCCESS

44.7.1.12 eCLD_ZllCommissionCommandNetworkJoinRouterRspCommandSend

```
PUBLIC teZCL_Status eCLD_ZllCommissionCommandNetworkJoinRouterRspCommandSend(
        ZPS_tsInterPanAddress psDestinationAddress,
        uint8 *pu8TransactionSequenceNumber,
        tsCLD_ZllCommission_NetworkJoinRouterRspCommandPayload
 *psPayload);
```

Description

This function is used to send a Network Join Router Response command to confirm that the local (Router) node is ready to join a network in reply to a received Network Join Router Request from a remote node. The command is sent as an inter-PAN message.

A pointer must be provided to a structure containing the data to be returned.

The specified Transaction Sequence Number (TSN) of the response must match the TSN of the corresponding request, as this will allow the response to be paired with the request at the destination.

Parameters

psDestinationAddress Pointer to stucture containing PAN ID and address information for target node

pu8TransactionSequenceNumber Pointer to location containing the Transaction Sequence Number (TSN) of the response

psPayload Pointer to structure containing payload data for the Network Join Router Response command (see Section 44.8.14)

Returns

E_ZCL_SUCCESS

44.7.1.13 eCLD_ZllCommissionCommandNetworkJoinEndDeviceReqCommandSend

```
teZCL_Status eCLD_ZllCommissionCommandNetworkJoinEndDeviceReqCommandSend(
        ZPS_tsInterPanAddress *psDestinationAddress,
        uint8 *pu8TransactionSequenceNumber,
        tsCLD_ZllCommission_NetworkJoinEndDeviceReqCommandPayload
 *psPayload);
```

Description

This function is used to send a Network Join End Device Request command to allow a detected End Device to join the created network. The command is sent as an inter-PAN message.

The function can be called once a network has been created.

The command payload contains information about the network and the local node, as well as certain data for the target node. This data includes a range of network addresses and a range of group IDs from which the target End Device can assign values to the other nodes - in this case, the End Device would typically be a remote control unit. This payload information is detailed in Section 44.8.15.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is 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

psDestinationAddress Pointer to stucture containing PAN ID and address information for target node

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

psPayload Pointer to structure containing payload data for the Network Join End Device Request command (see Section 44.8.15)

Returns

E_ZCL_SUCCESS

44.7.1.14 eCLD_ZllCommissionCommandNetworkJoinEndDeviceRspCommandSend

```
PUBLIC teZCL_Status eCLD_ZllCommissionCommandNetworkJoinEndDeviceRspCommandSend(
        ZPS_tsInterPanAddress *psDestinationAddress,
        uint8 *pu8TransactionSequenceNumber,
        tsCLD_ZllCommission_NetworkJoinEndDeviceRspCommandPayload
 *psPayload);
```

Description

This function is used to send a Network Join End Device Response command to confirm that the local (End Device) node is ready to join a network in reply to a received Network Join End Device Request from a remote node. The command is sent as an inter-PAN message.

A pointer must be provided to a structure containing the data to be returned.

The specified Transaction Sequence Number (TSN) of the response must match the TSN of the corresponding request, as this will allow the response to be paired with the request at the destination.

Parameters

psDestinationAddress Pointer to stucture containing PAN ID and address information for target node
pu8TransactionSequenceNumber Pointer to location containing the Transaction Sequence Number (TSN) of the response
psPayload Pointer to structure containing payload data for the Network Join End Device Response command (see Section 44.8.16)

Returns

E_ZCL_SUCCESS

44.7.1.15 eCLD_ZllCommissionCommandNetworkUpdateReqCommandSend

```
teZCL_Status         eCLD_ZllCommissionCommandNetworkUpdateReqCommandSend(
        ZPS_tsInterPanAddress *psDestinationAddress,
        uint8 *pu8TransactionSequenceNumber,
        tsCLD_ZllCommission_NetworkUpdateReqCommandPayload
 *psPayload);
```

Description

This function is used to send a Network Update Request command to bring a node that has missed a network update back into the network. The command is sent as an inter-PAN message.

The command payload contains information about the network, including the current value of the Network Update Identifier. This identifier takes a value in the range 0x00 to 0xFF and is incremented when a network

update has occurred (the value wraps around at 0xFF). Thus, if this value in the payload is more recent than the value of this identifier held by the target node, the node should update its network settings using the values in the rest of the payload. The payload information is detailed in Section 44.8.17.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is 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

psDestinationAddress Pointer to stucture containing PAN ID and address information for target node
pu8TransactionSequenceNumber Pointer to a location to store the Transaction Sequence Number (TSN) of the request
psPayload Pointer to structure containing payload data for the Network Update Request command (see Section 44.8.17)

Returns

E_ZCL_SUCCESS

44.7.2 Commissioning Utility functions

The following Commissioning Utility functions are provided:

1. eCLD_ZllUtilityCreateUtility
2. eCLD_ZllUtilityCommandEndpointInformationCommandSend
3. eCLD_ZllUtilityCommandGetGroupIdReqCommandSend
4. eCLD_ZllUtilityCommandGetGroupIdRspCommandSend
5. eCLD_ZllUtilityCommandGetEndpointListReqCommandSend
6. eCLD_ZllUtilityCommandGetEndpointListRspCommandSend
7. eCLD_ZllUtilityCommandHandler

44.7.2.1 eCLD_ZllUtilityCreateUtility

```
teZCL_Status eCLD_ZllUtilityCreateUtility(
        tsZCL_ClusterInstance *psClusterInstance,
        bool_t bIsServer,
        tsZCL_ClusterDefinition *psClusterDefinition,
        void *pvSharedStructPtr,
        tsZCL_AttributeStatus psAttributeStatus,
        tsCLD_ZllUtilityCustomDataStructure
 *psCustomDataStructure);
```

Description

This function creates a Touchlink Commissioning cluster instance for the Commissioning Utility. The cluster instance is created on the endpoint of the calling application, which should be the main application on the node. The type of cluster instance (server or client) to be created must be specified.

Parameters

psClusterInstance Pointer to cluster instance structure on local endpoint
bIsServer Type of cluster instance (server or client) to be created:

TRUE - server
FALSE - client
psClusterDefinition Pointer to cluster definition structure containing information about the cluster
pvSharedStructPtr Pointer to structure containing the shared storage for the cluster
psAttributeStatus Pointer to a structure containing the storage for each attribute's status
psCustomDataStructure Pointer to custom data to be provided to the cluster (see Section 44.8.20)

Returns

E_ZCL_SUCCESS

44.7.2.2 eCLD_ZllUtilityCommandEndpointInformationCommandSend

```
teZCL_Status eCLD_ZllUtilityCommandEndpointInformationCommandSend(
        uint8 u8SrcEndpoint,
        uint8 u8DstEndpoint,
        tsZCL_Address *psDestinationAddress,
        uint8 *pu8TransactionSequenceNumber,
        tsCLD_ZllUtility_EndpointInformationCommandPayload
 *psPayload);
```

Description

This function is used to send an Endpoint Information command to provide a remote endpoint with general information about the local endpoint (this may prompt the remote endpoint to request further information about the local endpoint). The function would typically be used to send local endpoint information from a ‘teacher’ node to a ‘learner’ node, in order to facilitate two-way communication between the Commissioning Utilities on the two nodes.

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

Parameters

u8SrcEndpoint Number of local endpoint (1-240)
u8DstEndpoint Number of destination endpoint (1-240)
psDestinationAddress Pointer to stucture containing address information for target node
pu8TransactionSequenceNumber Pointer to a location to store the Transaction Sequence Number (TSN) of the command
psPayload Pointer to structure to contain payload data for the Endpoint Information command (see Section 44.8.20)

Returns

E_ZCL_SUCCESS

44.7.2.3 eCLD_ZllUtilityCommandGetGroupIdReqCommandSend

```
teZCL_Status eCLD_ZllUtilityCommandGetGroupIdReqCommandSend(
        uint8 u8Srcendpoint,
```

```
        uint8 u8DstEndpoint,
        tsZCL_Address *psDestinationAddress,
        uint8 *pu8TransactionSequenceNumber,
        uint8 u8StartIndex);
```

Description

This function is used to send a Get Group Identifiers Request command to obtain information about the groups (of lights) that have been configured on a remote endpoint. The function would typically be used on a ‘learner’ node to request the groups that have been configured on a ‘teacher’ node.

The first group from the groups list to be included in the returned information must be specified in terms of an index.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is 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

u8SrcEndpoint Number of local endpoint (1-240)
u8DstEndpoint Number of destination endpoint (1-240)
psDestinationAddress Pointer to stucture containing address information for target node
pu8TransactionSequenceNumber Pointer to a location to store the Transaction Sequence Number (TSN) of the request
u8StartIndex Index in group list of the first group to include in the returned information

Returns

E_ZCL_SUCCESS

44.7.2.4 eCLD_ZllUtilityCommandGetGroupIdRspCommandSend

```
PUBLIC teZCL_Status eCLD_ZllUtilityCommandGetGroupIdRspCommandSend(
        uint8 u8SrcEndpoint,
        uint8 u8DstEndpoint,
        tsZCL_Address *psDestinationAddress,
        uint8 *pu8TransactionSequenceNumber,
        uint8 u8StartIndex);
```

Description

This function is used to send a Get Group Identifiers Response command containing information about the groups (of lights) that have been configured on the local endpoint. The function would typically be used on a ‘teacher’ node to respond to a Get Group Identifiers Request from a ‘learner’ node.

The first group from the groups list to be included in the returned information must be specified in terms of an index. The returned information includes this index, the number of (consecutive) groups included and the identifier of each group.

The specified Transaction Sequence Number (TSN) of the response must match the TSN of the corresponding request, as this will allow the response to be paired with the request at the destination.

Parameters

u8SrcEndpoint Number of local endpoint (1-240)
u8DstEndpoint Number of destination endpoint (1-240)
psDestinationAddress Pointer to stucture containing address information for target node
pu8TransactionSequenceNumber Pointer to location containing the Transaction Sequence Number (TSN) of the response
u8StartIndex Index in group list of the first group to include in the returned information

Returns

E_ZCL_SUCCESS

44.7.2.5 eCLD_ZllUtilityCommandGetEndpointListReqCommandSend

```
teZCL_Status eCLD_ZllUtilityCommandGetEndpointListReqCommandSend(
        uint8 u8SrcEndpoint,
        uint8 u8DstEndpoint,
        tsZCL_Address *psDestinationAddress,
        uint8 *pu8TransactionSequenceNumber,
        uint8 u8StartIndex);
```

Description

This function is used to send a Get Endpoint List Request command to obtain information about controlled endpoints. The function would typically be used on a ‘learner’ node to request the remote endpoints that a ‘teacher’ node controls.

The first endpoint from the endpoints list to be included in the returned information must be specified in terms of an index.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the request. The TSN in the response is 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

u8SrcEndpoint Number of local endpoint (1-240)
u8DstEndpoint Number of destination endpoint (1-240)
psDestinationAddress Pointer to stucture containing address information for target node
pu8TransactionSequenceNumber Pointer to a location to store the Transaction Sequence Number (TSN) of the request
u8StartIndex Index in endpoint list of the first endpoint to include in the returned information

Returns

E_ZCL_SUCCESS

44.7.2.6 eCLD_ZllUtilityCommandGetEndpointListRspCommandSend

```
PUBLIC teZCL_Status eCLD_ZllUtilityCommandGetEndpointListRspCommandSend(
        uint8 u8SrcEndpoint,
        uint8 u8DstEndpoint,
```

```
        tsZCL_Address *psDestinationAddress,
        uint8 *pu8TransactionSequenceNumber,
        uint8 u8StartIndex);
```

Description

This function is used to send a Get Endpoint List Response command containing information about controlled endpoints. The function would typically be used on a ‘teacher’ node to respond to a Get Endpoint List Request from a ‘learner’ node.

The first endpoint from the endpoints list to be included in the returned information must be specified in terms of an index. The returned information will include this index, the number of (consecutive) endpoints included and the information about each endpoint (including endpoint number, identifier of resident ZigBee device and version of this device).

The specified Transaction Sequence Number (TSN) of the response must match the TSN of the corresponding request, as this will allow the response to be paired with the request at the destination.

Parameters

u8SrcEndpoint Number of local endpoint (1-240)
u8DstEndpoint Number of destination endpoint (1-240)
psDestinationAddress Pointer to stucture containing address information for target node
pu8TransactionSequenceNumber Pointer to location containing the Transaction Sequence Number (TSN) of the response
u8StartIndex Index in endpoint list of the first endpoint to include in the returned information

Returns

E_ZCL_SUCCESS

44.7.2.7 eCLD_ZllUtilityCommandHandler

```
teZCL_Status eCLD_ZllUtilityCommandHandler(
        ZPS_tsAfEvent *pZPSevent,
        tsZCL_EndPointDefinition *psEndPointDefinition,
        tsZCL_ClusterInstance         *psClusterInstance);
```

Description

This function parses a ZigBee PRO event and invokes the user-defined callback function that has been registered for the device (using the relevant endpoint registration function).

The registered user-defined callback function must be designed to handle events associated with the Commissioning Utility.

Parameters

pZPSevent Pointer to received ZigBee PRO event
psEndPointDefinition Pointer to structure which defines endpoint on which the Commissioning Utility resides
psClusterInstance Pointer to Touchlink Commissioning cluster instance structure

Returns

E_ZCL_SUCCESS

44.8 Structures

This section details the structures used in the Touchlink Commissioning cluster (both Touchlink and Commissioning Utility parts).

44.8.1 tsZLL_CommissionEndpoint

This structure is used to hold endpoint information for a Touchlink application.

```
typedef struct
{
tsZCL_EndPointDefinition sEndPoint;
tsZLL_CommissionEndpointClusterInstances sClusterInstance;
#if (defined CLD_ZLL_COMMISSION) && (defined ZLL_COMMISSION_SERVER)
    tsCLD_ZllCommission sZllCommissionServerCluster;
    tsCLD_ZllCommissionCustomDataStructure
                sZllCommissionServerCustomDataStructure;
#endif
#if (defined CLD_ZLL_COMMISSION) && (defined ZLL_COMMISSION_CLIENT)
    tsCLD_ZllCommission sZllCommissionClientCluster;
    tsCLD_ZllCommissionCustomDataStructure
              sZllCommissionClientCustomDataStructure;
#endif
} tsZLL_CommissionEndpoint;
```

where:

- `sEndPoint` is a ZCL structure containing information about the endpoint (refer to Section 6.1.1).
- `sClusterInstance` is a structure containing information about the Touchlink Commissioning cluster instance on the endpoint (see Section 44.8.2).
- For a Touchlink server, the following fields are used:
  - `sZllCommissionServerCluster` is the Touchlink Commissioning cluster structure (which contains no attributes).
  - `sZllCommissionServerCustomDataStructure` is a structure containing custom data for the cluster server (see Section 44.8.3).
- For a Touchlink client, the following fields are used:
  - `sZllCommissionClientCluster` is the Touchlink Commissioning cluster structure (which contains no attributes).
  - `sZllCommissionClientCustomDataStructure` is a structure containing custom data for the cluster client (see Section 44.8.3).

44.8.2 tsZLL_CommissionEndpointClusterInstances

This structure holds information about the Touchlink Commissioning cluster instance on an endpoint.

```
typedef struct PACK
{
#if (defined CLD_ZLL_COMMISSION) && (defined ZLL_COMMISSION_SERVER)
    tsZCL_ClusterInstance sZllCommissionServer;
#endif
#if (defined CLD_ZLL_COMMISSION) && (defined ZLL_COMMISSION_CLIENT)
```

```
    tsZCL_ClusterInstance sZllCommissionClient;
#endif
} tsZLL_CommissionEndpointClusterInstances;
```

where:

- `sZllCommissionServer` is a ZCL structure containing information about the Touchlink Commissioning cluster server instance (refer to Section 6.1.16).
- `sZllCommissionClient` is a ZCL structure containing information about the Touchlink Commissioning cluster client instance (refer to Section 6.1.16).

44.8.3 tsCLD_ZllCommissionCustomDataStructure

This structure is used to hold the data for a Touchlink command received by a node.

```
typedef struct
{
    tsZCL_ReceiveEventAddressInterPan    sRxInterPanAddr;
    tsZCL_CallBackEvent                  sCustomCallBackEvent;
    tsCLD_ZllCommissionCallBackMessage   sCallBackMessage;
} tsCLD_ZllCommissionCustomDataStructure;
```

where:

- `RxInterPanAddr` is a ZCL structure containing the Inter-PAN addresses of the source and destination nodes of the command.
- `sCustomCallBackEvent` is the ZCL event structure for the command.
- `sCallBackMessage` is a structure containing the command ID and payload (see Section 44.8.4).

44.8.4 tsCLD_ZllCommissionCallBackMessage

This structure contains the command ID and payload for a received Touchlink command.

```
typedef struct
{
    uint8   u8CommandId;
    union
    {
        tsCLD_ZllCommission_ScanReqCommandPayload
                                    *psScanReqPayload;
        tsCLD_ZllCommission_ScanRspCommandPayload
                                    *psScanRspPayload;
        tsCLD_ZllCommission_IdentifyReqCommandPayload
                                    *psIdentifyReqPayload;
        tsCLD_ZllCommission_DeviceInfoReqCommandPayload
                                    *psDeviceInfoReqPayload;
        tsCLD_ZllCommission_DeviceInfoRspCommandPayload
                                    *psDeviceInfoRspPayload;
        tsCLD_ZllCommission_FactoryResetReqCommandPayload
                                    *psFactoryResetPayload;
        tsCLD_ZllCommission_NetworkStartReqCommandPayload
                                    *psNwkStartReqPayload;
        tsCLD_ZllCommission_NetworkStartRspCommandPayload
                                    *psNwkStartRspPayload;
        tsCLD_ZllCommission_NetworkJoinRouterReqCommandPayload
                                    *psNwkJoinRouterReqPayload;
        tsCLD_ZllCommission_NetworkJoinRouterRspCommandPayload
                                    *psNwkJoinRouterRspPayload;
```

```
        tsCLD_ZllCommission_NetworkJoinEndDeviceReqCommandPayload
                                    *psNwkJoinEndDeviceReqPayload;
        tsCLD_ZllCommission_NetworkJoinEndDeviceRspCommandPayload
                                    *psNwkJoinEndDeviceRspPayload;
        tsCLD_ZllCommission_NetworkUpdateReqCommandPayload
                                    *psNwkUpdateReqPayload;
    } uMessage;
} tsCLD_ZllCommissionCallBackMessage;
```

where:

- `u8CommandId` is the command ID - enumerations are provided, as detailed in Section 44.6.1.
- `uMessage` contains the payload of the command, where the structure used depends on the command ID (the structures are detailed in the sections below).

44.8.5 tsCLD_ZllCommission_ScanReqCommandPayload

This structure is used to hold the payload data for a Touchlink Scan Request command.

```
typedef struct
{
    uint32 u32TransactionId;
    uint8  u8ZigbeeInfo;
    uint8  u8ZllInfo;
} tsCLD_ZllCommission_ScanReqCommandPayload;
```

where:

- `u32TransactionId` is the 32-bit Inter-PAN Transaction Identifier of the request. This is a random number generated and inserted automatically.
- `u8ZigbeeInfo` is a bitmap of ZigBee information which indicates the ZigBee device type of the sending node and whether the radio receiver remains on when the node is idle. This information is inserted by the ZigBee stack.
- `u8ZllInfo` is a bitmap indicating properties of the sending node, including whether the node is factory new, whether the node is able to assign addresses to other nodes and whether the node is able to initiate a link operation (supports Touchlink Commissioning cluster on the client side). This information is inserted automatically.

44.8.6 tsCLD_ZllCommission_ScanRspCommandPayload

This structure is used to hold the payload data for a Touchlink Scan Response command.

```
typedef struct
{
    uint32  u32TransactionId;
    uint8   u8RSSICorrection;
    uint8   u8ZigbeeInfo;
    uint8   u8ZllInfo;
    uint16  u16KeyMask;
    uint32  u32ResponseId;
    uint64  u64ExtPanId;
    uint8   u8NwkUpdateId;
    uint8   u8LogicalChannel;
    uint16  u16PanId;
    uint16  u16NwkAddr;
    uint8   u8NumberSubDevices;
    uint8   u8TotalGroupIds;
```

```
    uint8   u8Endpoint;
    uint16  u16ProfileId;
    uint16  u16DeviceId;
    uint8   u8Version;
    uint8   u8GroupIdCount;
} tsCLD_ZllCommission_ScanRspCommandPayload;
```

where:

- `u32TransactionId` is the 32-bit Inter-PAN Transaction Identifier of the response, which must take the same value as the identifier in the corresponding request.
- `u8RSSICorrection` is the 8-bit RSSI correction offset for the node, in the range 0x00 to 0x20.
- `u8ZigbeeInfo` is an 8-bit field containing the following ZigBee-related information:
  - Bits 1-0: Node type (00 - Co-ordinator, 01 - Router, 10 - End Device)
  - Bit 2: Rx on when idle (1 - On, 0 - Off)
  - Bits 7-3: Reserved
- `u8ZllInfo` is an 8-bit field containing the following information:
  - Bit 0: Factory new (1 - Yes, 0 - No)
  - Bit 1: Address assignment capability (1 - Yes, 0 - No)
  - Bits 3-2: Reserved
  - Bit 4: Touchlink initiator (1 - Yes, 0 - No)
  - Bit 5: Touchlink priority request (1 - Yes, 0 - No)
  - Bits 7-6: Reserved
- `u16KeyMask` is a 16-bit bitmap indicating which link key is installed on the node - only one bit should be set to '1', corresponding to the key that is in use. The possible values and keys are:
  - 0x0001 (bit 0 set): Development key (defined by developer for use during application development)
  - 0x0010 (bit 4 set): Master key (obtained from the ZigBee Alliance after successful certification and agreement with the terms of the 'ZLL Security Key Licence and Confidentialty Agreement')
  - 0x8000 (bit 15 set): Certification key (defined in the ZLL Specification for use during development and during certification at test houses)
- `u32ResponseId` is a 32-bit random identifier for the response, used during network key transfer.
- `u64ExtPanId` is the 64-bit Extended PAN ID of a network to which the node already belongs, if any (a zero value indicates no network membership).
- `u8NwkUpdateId` is the current value of the Network Update Identifier on the node (see Section 44.4.3).
- `u8LogicalChannel` is the number of the IEEE 802.15.4 radio channel used by a network to which the node already belongs, if any (a zero value indicates no network membership and therefore that no particular channel is used).
- `u16PanId` is the 16-bit PAN ID of a network to which the node already belongs, if any (a zero value indicates no network membership).
- `u16NwkAddr` is the 16-bit network address currently assigned to the node (the value 0xFFFF indicates that the node is 'factory new' and has no assigned network address).
- `u8NumberSubDevices` is the number of ZigBee devices on the node.
- `u8TotalGroupIds` is the total number of groups (of lights) supported on the node (across all devices).
- `u8Endpoint` is number of the endpoint (in the range 1-240) on which the ZigBee device is resident (this field is only used when there is only one ZigBee device on the node).
- `u16ProfileId` is the 16-bit identifier of the ZigBee application profile that is supported by the device (this field is only used when there is only one ZigBee device on the node).
- `u16DeviceId` is the 16-bit Device Identifier supported by the device (this field is only used when there is only one ZigBee device on the node).

- `u8Version` is an 8-bit version number for the device - the four least significant bits are from the Application Device Version field of the appropriate Simple Descriptor and the four most significant bits are zero (this field is only used when there is only one ZigBee device on the node).
- `u8GroupIdCount` is the number of groups (of lights) supported by the device (this field is only used when there is only one ZigBee device on the node).

44.8.7 tsCLD_ZllCommission_DeviceInfoReqCommandPayload

This structure is used to hold the payload data for a Touchlink Device Information Request command.

```
typedef struct
{
    uint32 u32TransactionId;
    uint8  u8StartIndex;
} tsCLD_ZllCommission_DeviceInfoReqCommandPayload;
```

where:

- `u32TransactionId` is the 32-bit Inter-PAN Transaction Identifier of the request. This is a random number generated and inserted automatically.
- `u8StartIndex` specifies the index (starting from 0) of the first entry in the device table from which device information should be obtained.

44.8.8 tsCLD_ZllCommission_DeviceInfoRspCommandPayload

This structure is used to hold the payload data for a Touchlink Device Information Response command.

```
typedef struct
{
    uint32  u32TransactionId;
    uint8   u8NumberSubDevices;
    uint8   u8StartIndex;
    uint8   u8DeviceInfoRecordCount;
    tsCLD_ZllDeviceRecord asDeviceRecords[ZLL_MAX_DEVICE_RECORDS];
} tsCLD_ZllCommission_DeviceInfoRspCommandPayload;
```

where:

- `u32TransactionId` is the 32-bit Inter-PAN Transaction Identifier of the response, which must take the same value as the identifier in the corresponding request.
- `u8NumberSubDevices` is the number of ZigBee devices on the node (as reported in the Scan Response).
- `u8StartIndex` is the index (starting from 0) of the first entry in the device table from which device information has been obtained (this value should be as specified in the corresponding request).
- `u8DeviceInfoRecordCount` indicates the number of device information records included in the response (in the range 0 to 5).
- `asDeviceRecords[]` is an array, where each array element is a `tsCLD_ZllDeviceRecord` structure containing a device information record for one ZigBee device on the node.

44.8.9 tsCLD_ZllCommission_IdentifyReqCommandPayload

This structure is used to hold the payload data for a Touchlink Identify Request command.

```
typedef struct
{
    uint32  u32TransactionId;
```

```
    uint16  u16Duration;
} tsCLD_ZllCommission_IdentifyReqCommandPayload;
```

where:

- `u32TransactionId` is the 32-bit Inter-PAN Transaction Identifier of the request. This is a random number generated and inserted automatically.
- `u16Duration` specifies the length of time (in seconds) that the target node is to remain in identify mode. The possible values are:
  - 0x0000: Exit identify mode immediately
  - 0x0001–0xFFFE: Number of seconds to remain in identify mode
  - 0xFFFF: Remain in identify mode for the default time for the target node

    If the target node is unable to provide accurate timings, it will attempt to remain in identify mode for as close to the requested time as possible

44.8.10 tsCLD_ZllCommission_FactoryResetReqCommandPayload

This structure is used to hold the payload data for a Touchlink Reset to Factory New Request command.

```
typedef struct
{
    uint32 u32TransactionId;
} tsCLD_ZllCommission_FactoryResetReqCommandPayload;
```

where `u32TransactionId` is the 32-bit Inter-PAN Transaction Identifier of the request. This is a random number generated and inserted automatically.

44.8.11 tsCLD_ZllCommission_NetworkStartReqCommandPayload

This structure is used to hold the payload data for a Touchlink Network Start Request command.

```
typedef struct
{
    uint32  u32TransactionId;
    uint64  u64ExtPanId;
    uint8   u8KeyIndex;
    uint8   au8NwkKey[16];
    uint8   u8LogicalChannel;
    uint16  u16PanId;
    uint16  u16NwkAddr;
    uint16  u16GroupIdBegin;
    uint16  u16GroupIdEnd;
    uint16  u16FreeNwkAddrBegin;
    uint16  u16FreeNwkAddrEnd;
    uint16  u16FreeGroupIdBegin;
    uint16  u16FreeGroupIdEnd;
    uint64  u64InitiatorIEEEAddr;
    uint16  u16InitiatorNwkAddr;
} tsCLD_ZllCommission_NetworkStartReqCommandPayload;
```

where:

- `u32TransactionId` is the 32-bit Inter-PAN Transaction Identifier of the request. This is a random number generated and inserted automatically.

- u64ExtPanId is the Extended PAN ID (EPID) of the new network (if set to zero, the target node will choose the EPID).
- u8KeyIndex is a value indicating the type of security key used to encrypt the randomly generated network key in au8NwkKey. The valid values are as follows (all other values are reserved for future use):
  - 0: Development key, used during development before ZigBee certification
  - 4: Master key, used after successful ZigBee certification
  - 15: Certification key, used during ZigBee certification testing
- au8NwkKey[16] is the 128-bit randomly generated network key encrypted using the key specified in u8KeyIndex.
- u8LogicalChannel is the number of the IEEE 802.15.4 radio channel to be used by the network (if set to zero, the target node will choose the channel).
- u16PanId is the PAN ID of the new network (if set to zero, the target node will choose the PAN ID).
- u16NwkAddr is the 16-bit network (short) address assigned to the target node
- u16GroupIdBegin is the start value of the range of group identifiers that the target node can use for its own endpoints (if set to zero, no range of group identifiers has been allocated).
- u16GroupIdEnd is the end value of the range of group identifiers that the target node can use for its own endpoints (if set to zero, no range of group identifiers has been allocated).
- u16FreeNwkAddrBegin is the start address of the range of network addresses that the target node can assign to other nodes (if set to zero, no range of network addresses has been allocated).
- u16FreeNwkAddrEnd is the end address of the range of network addresses that the target node can assign to other nodes (if set to zero, no range of network addresses has been allocated).
- u16FreeGroupIdBegin is the start value of the range of free group identifiers that the target node can assign to other nodes (if set to zero, no range of free group identifiers has been allocated).
- u16FreeGroupIdEnd is the end value of the range of free group identifiers that the target node can assign to other nodes (if set to zero, no range of free group identifiers has been allocated).
- u64InitiatorIEEEAddr is the IEEE (MAC) address of the local node (network initiator)
- u16InitiatorNwkAddr is the network (short) address of the local node (network initiator)

44.8.12 tsCLD_ZllCommission_NetworkStartRspCommandPayload

This structure is used to hold the payload data for a Touchlink Network Start Response command.

```
typedef struct
{
    uint32  u32TransactionId;
    uint8   u8Status;
    uint64  u64ExtPanId;
    uint8   u8NwkUpdateId;
    uint8   u8LogicalChannel;
    uint16  u16PanId;
} tsCLD_ZllCommission_NetworkStartRspCommandPayload;
```

where:

- u32TransactionId is the 32-bit Inter-PAN Transaction Identifier of the response, which must take the same value as the identifier in the corresponding request.
- u8Status indicates the outcome of the corresponding Network Start Request: 0x00 for success, 0x01 for failure.
- u64ExtPanId is the Extended PAN ID (EPID) of the new network (this will be the value specified in the corresponding request or a value chosen by the local node).
- u8NwkUpdateId is the current value of the Network Update Identifier, which will be set to zero for a new network (see Section 44.4.3).

- u8LogicalChannel is the number of the IEEE 802.15.4 radio channel to be used by the network (this will be the value specified in the corresponding request or a value chosen by the local node).
- u16PanId is the PAN ID of the new network (this will be the value specified in the corresponding request or a value chosen by the local node).

44.8.13 tsCLD_ZllCommission_NetworkJoinRouterReqCommandPayload

This structure is used to hold the payload data for a Touchlink Network Join Router Request command.

```
typedef struct
{
    uint32  u32TransactionId;
    uint64  u64ExtPanId;
    uint8   u8KeyIndex;
    uint8   au8NwkKey[16];
    uint8   u8NwkUpdateId;
    uint8   u8LogicalChannel;
    uint16  u16PanId;
    uint16  u16NwkAddr;
    uint16  u16GroupIdBegin;
    uint16  u16GroupIdEnd;
    uint16  u16FreeNwkAddrBegin;
    uint16  u16FreeNwkAddrEnd;
    uint16  u16FreeGroupIdBegin;
    uint16  u16FreeGroupIdEnd;
} tsCLD_ZllCommission_NetworkJoinRouterReqCommandPayload;
```

where:

- u32TransactionId is the 32-bit Inter-PAN Transaction Identifier of the request. This is a random number generated and inserted automatically.
- u64ExtPanId is the Extended PAN ID (EPID) of the network.
- u8KeyIndex is a value indicating the type of security key used to encrypt the network key in au8NwkKey. The valid values are as follows (all other values are reserved for future use):
  - 0: Development key, used during development before ZigBee certification
  - 4: Master key, used after successful ZigBee certification
  - 15: Certification key, used during ZigBee certification testing
- au8NwkKey[16] is the 128-bit network key encrypted using the key specified in u8KeyIndex.
- u8NwkUpdateId is the current value of the Network Update Identifier. This identifier takes a value in the range 0x00 to 0xFF and is incremented when a network update has occurred which requires the network settings on the nodes to be changed.
- u8LogicalChannel is the number of the IEEE 802.15.4 radio channel used by the network.
- u16PanId is the PAN ID of the network
- u16NwkAddr is the 16-bit network (short) address assigned to the target node
- u16GroupIdBegin is the start value of the range of group identifiers that the target node can use for its own endpoints (if set to zero, no range of group identifiers has been allocated).
- u16GroupIdEnd is the end value of the range of group identifiers that the target node can use for its own endpoints (if set to zero, no range of group identifiers has been allocated).
- u16FreeNwkAddrBegin is the start address of the range of network addresses that the target node can assign to other nodes (if set to zero, no range of network addresses has been allocated).
- u16FreeNwkAddrEnd is the end address of the range of network addresses that the target node can assign to other nodes (if set to zero, no range of network addresses has been allocated).
- u16FreeGroupIdBegin is the start value of the range of free group identifiers that the target node can assign to other nodes (if set to zero, no range of free group identifiers has been allocated).

- u16FreeGroupIdEnd is the end value of the range of free group identifiers that the target node can assign to other nodes (if set to zero, no range of free group identifiers has been allocated).

44.8.14 tsCLD_ZllCommission_NetworkJoinRouterRspCommandPayload

This structure is used to hold the payload data for a Touchlink Network Join Router Response command.

```
typedef struct
{
    uint32  u32TransactionId;
    uint8   u8Status;
} tsCLD_ZllCommission_NetworkJoinRouterRspCommandPayload;
```

where:

- u32TransactionId is the 32-bit Inter-PAN Transaction Identifier of the response, which must take the same value as the identifier in the corresponding request.
- u8Status indicates the outcome of the corresponding Network Join Router Request: 0x00 for success, 0x01 for failure.

44.8.15 tsCLD_ZllCommission_NetworkJoinEndDeviceReqCommandPayload

This structure is used to hold the payload data for a Touchlink Network Join End Device Request command.

```
typedef struct
{
    uint32  u32TransactionId;
    uint64  u64ExtPanId;
    uint8   u8KeyIndex;
    uint8   au8NwkKey[16];
    uint8   u8NwkUpdateId;
    uint8   u8LogicalChannel;
    uint16  u16PanId;
    uint16  u16NwkAddr;
    uint16  u16GroupIdBegin;
    uint16  u16GroupIdEnd;
    uint16  u16FreeNwkAddrBegin;
    uint16  u16FreeNwkAddrEnd;
    uint16  u16FreeGroupIdBegin;
    uint16  u16FreeGroupIdEnd;
} tsCLD_ZllCommission_NetworkJoinEndDeviceReqCommandPayload;
```

where:

- u32TransactionId is the 32-bit Inter-PAN Transaction Identifier of the request. This is a random number generated and inserted automatically.
- u64ExtPanId is the Extended PAN ID (EPID) of the network.
- u8KeyIndex is a value indicating the type of security key used to encrypt the network key in au8NwkKey. The valid values are as follows (all other values are reserved for future use):
  - 0: Development key, used during development before ZigBee certification
  - 4: Master key, used after successful ZigBee certification
  - 15: Certification key, used during ZigBee certification testing
- au8NwkKey[16] is the 128-bit network key encrypted using the key specified in u8KeyIndex.
- u8NwkUpdateId is the current value of the Network Update Identifier. This identifier takes a value in the range 0x00 to 0xFF and is incremented when a network update has occurred which requires the network settings on the nodes to be changed.

- u8LogicalChannel is the number of the IEEE 802.15.4 radio channel used by the network.
- u16PanId is the PAN ID of the network.
- u16NwkAddr is the 16-bit network (short) address assigned to the target node.
- u16GroupIdBegin is the start value of the range of group identifiers that the target node can use for its own endpoints (if set to zero, no range of group identifiers has been allocated).
- u16GroupIdEnd is the end value of the range of group identifiers that the target node can use for its own endpoints (if set to zero, no range of group identifiers has been allocated).
- u16FreeNwkAddrBegin is the start address of the range of network addresses that the target node can assign to other nodes (if set to zero, no range of network addresses has been allocated).
- u16FreeNwkAddrEnd is the end address of the range of network addresses that the target node can assign to other nodes (if set to zero, no range of network addresses has been allocated).
- u16FreeGroupIdBegin is the start value of the range of free group identifiers that the target node can assign to other nodes (if set to zero, no range of free group identifiers has been allocated).
- u16FreeGroupIdEnd is the end value of the range of free group identifiers that the target node can assign to other nodes (if set to zero, no range of free group identifiers has been allocated).

44.8.16 tsCLD_ZllCommission_NetworkJoinEndDeviceRspCommandPayload

This structure is used to hold the payload data for a Touchlink Network Join End Device Response command.

```
typedef struct
{
    uint32 u32TransactionId;
    uint8   u8Status;
} tsCLD_ZllCommission_NetworkJoinEndDeviceRspCommandPayload;
```

where:

- u32TransactionId is the 32-bit Inter-PAN Transaction Identifier of the response, which must take the same value as the identifier in the corresponding request.
- u8Status indicates the outcome of the corresponding Network Join End Device Request: 0x00 for success, 0x01 for failure.

44.8.17 tsCLD_ZllCommission_NetworkUpdateReqCommandPayload

This structure is used to hold the payload data for a Touchlink Network Update Request command.

```
typedef struct
{
 uint32  u32TransactionId;
 uint64  u64ExtPanId;
 uint8u8NwkUpdateId;
 uint8u8LogicalChannel;
 uint16  u16PanId;
 uint16  u16NwkAddr;
} tsCLD_ZllCommission_NetworkUpdateReqCommandPayload;
```

where:

- u32TransactionId is the 32-bit Inter-PAN Transaction Identifier of the request. This is a random number generated and inserted automatically.
- u64ExtPanId is the Extended PAN ID (EPID) of the network.
- u8NwkUpdateId is the current value of the Network Update Identifier (see Section 44.4.3).
- u8LogicalChannel is the number of the IEEE 802.15.4 radio channel used by the network.

- u16PanId is the PAN ID of the network.
- u16NwkAddr is the 16-bit network (short) address assigned to the target node.

44.8.18 tsCLD_ZllUtilityCustomDataStructure

This structure is used to hold custom data for a Commissioning Utility instance of the Touchlink Commissioning cluster.

```
typedef struct
{
 tsZCL_ReceiveEventAddress sRxAddr;
 tsZCL_CallBackEvent sCustomCallBackEvent;
 tsCLD_ZllUtilityCallBackMessage sCallBackMessage;
} tsCLD_ZllUtilityCustomDataStructure;
```

where:

- sRxAddr is a ZCL structure containing the destination address of the command.
- sCustomCallBackEvent is the ZCL event structure for the command.
- sCallBackMessage is a structure containing the command ID and payload (see Section 44.8.19).

44.8.19 tsCLD_ZllUtilityCallBackMessage

This structure contains the command ID and payload for a received Commissioning Utility command.

```
typedef struct
{
 uint8u8CommandId;
 union
 {
  tsCLD_ZllUtility_EndpointInformationCommandPayload
 *psEndpointInfoPayload;
  tsCLD_ZllUtility_GetGroupIdReqCommandPayload
 *psGetGroupIdReqPayload;
  tsCLD_ZllUtility_GetGroupIdRspCommandPayload
 *psGetGroupIdRspPayload;
  tsCLD_ZllUtility_GetEndpointListReqCommandPayload
 *psGetEndpointlistReqPayload;
  tsCLD_ZllUtility_GetEndpointListRspCommandPayload
 *psGetEndpointListRspPayload;
 } uMessage;
} tsCLD_ZllUtilityCallBackMessage;
```

where:

- u8CommandId is the command ID - enumerations are provided, as detailed in Section 44.6.2.
- uMessage contains the payload of the command, where the structure used depends on the command ID (the structures are detailed in the sections below).

44.8.20 tsCLD_ZllUtility_EndpointInformationCommandPayload

This structure is used to hold the payload data for a Commissioning Utility Endpoint Information command.

```
typedef struct
{
 uint64  u64IEEEAddr;
 uint16  u16NwkAddr;
```

```
 uint8u8Endpoint;
 uint16  u16ProfileID;
 uint16  u16DeviceID;
 uint8u8Version;
} tsCLD_ZllUtility_EndpointInformationCommandPayload;
```

where:

- `u64IEEEAddr` is the IEEE (MAC) address of the local node.
- `u16NwkAddr` is the network (short) address of the local node.
- `u8Endpoint` is the number of the local endpoint (1-240).
- `u16ProfileID` is the identifier of the ZigBee application profile supported on the local endpoint.
- `u16DeviceID` is identifier of the ZigBee device on the local endpoint.
- `u8Version` specifies the version number of the ZigBee device on the local endpoint.

44.9 Enumerations

44.9.1 Touchlink event enumerations

The event types generated by the Touchlink part of the Touchlink Commissioning cluster are enumerated in the `teCLD_ZllCommission_Command` structure below:

```
typedef enum PACK
{
   E_CLD_COMMISSION_CMD_SCAN_REQ  0x00,
   E_CLD_COMMISSION_CMD_SCAN_RSP,
   E_CLD_COMMISSION_CMD_DEVICE_INFO_REQ,
   E_CLD_COMMISSION_CMD_DEVICE_INFO_RSP,
   E_CLD_COMMISSION_CMD_IDENTIFY_REQ  0x06,
   E_CLD_COMMISSION_CMD_FACTORY_RESET_REQ,
   E_CLD_COMMISSION_CMD_NETWORK_START_REQ  0x10,
   E_CLD_COMMISSION_CMD_NETWORK_START_RSP,
   E_CLD_COMMISSION_CMD_NETWORK_JOIN_ROUTER_REQ,
   E_CLD_COMMISSION_CMD_NETWORK_JOIN_ROUTER_RSP,
   E_CLD_COMMISSION_CMD_NETWORK_JOIN_END_DEVICE_REQ,
   E_CLD_COMMISSION_CMD_NETWORK_JOIN_END_DEVICE_RSP,
   E_CLD_COMMISSION_CMD_NETWORK_UPDATE_REQ,
} teCLD_ZllCommission_Command;
```

44.9.2 Commissioning utility event enumerations

The event types generated by the Commissioning Utility part of the Touchlink Commissioning cluster are enumerated in the `teCLD_ZllUtility_Command` structure below:

```
 typedef enum PACK
{
  E_CLD_UTILITY_CMD_ENDPOINT_INFO = 0x40,
  E_CLD_UTILITY_CMD_GET_GROUP_ID_REQ_RSP,
  E_CLD_UTILITY_CMD_GET_ENDPOINT_LIST_REQ_RSP,
} teCLD_ZllUtility_Command;
```

44.10 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 Touchlink Commissioning cluster.

The Touchlink Commissioning cluster is enabled as follows:

- **Touchlink -** To enable the cluster, define CLD_ZLL_COMMISSION, then:
  - to enable the cluster as a server, define ZLL_COMMISSION_SERVER
  - to enable the cluster as a client, define ZLL_COMMISSION_CLIENT
- **Commissioning Utility -** To enable the cluster, define CLD_ZLL_UTILITY, then:
  - to enable the cluster as a server, define ZLL_UTILITY_SERVER
  - to enable the cluster as a client, define ZLL_UTILITY_CLIENT

Part XI: Appliances Clusters

This part comprises four chapters:

- Chapter 45 details the **Appliance Control** cluster
- Chapter 46 details the **Appliance Identification** cluster
- Chapter 47 details the **Appliance Events and Alerts** cluster
- Chapter 48 details the **Appliance Statistics** cluster

45 Appliance Control Cluster

This chapter outlines the Appliance Control cluster which provides an interface for remotely controlling appliances in the home.

The Appliance Control cluster has a Cluster ID of 0x001B.

45.1 Overview

The Appliance Control cluster provides an interface for the remote control and programming of home appliances (e.g. a washing machine) by sending basic operational commands such as start, pause, stop.

The cluster is enabled by defining CLD_APPLIANCE_CONTROL in the **zcl_options.h** file. Further compile-time options for the Appliance Control cluster are detailed in Section 45.10.

All attributes of the Appliance Control cluster are in the 'Appliance Functions' attribute set.

45.2 Cluster structure and attributes

The structure definition for the Appliance Control cluster (server) is:

```
typedef struct
{
#ifdef APPLIANCE_CONTROL_SERVER
    zuint16                         u16StartTime;
    zuint16                         u16FinishTime;
#ifdef CLD_APPLIANCE_CONTROL_REMAINING_TIME
    zuint16                         u16RemainingTime;
#endif
#ifdef CLD_APPLIANCE_CONTROL_ATTRIBUTE_REPORTING_STATUS
    zenum8                          u8AttributeReportingStatus;
#endif
#endif
    zuint16                         u16ClusterRevision;
} tsCLD_ApplianceControl;
```

where:

- `u16StartTime` is a bitmap representing the start-time of a 'running' cycle of the appliance, as follows:

Bits	Description
0-5	Minutes part of the start-time, in the range 0 to 59 (may be absolute or relative time - see below)
6-7	Type of time encoding: • 0x0: Relative time - start-time is a delay from the time that the attribute was set • 0x1: Absolute time - start-time is an actual time of the 24-hour clock • 0x2-0x3: Reserved The defaults are absolute time for ovens and relative time for other appliances.
8-15	Hours part of the start-time: • in the range 0 to 255, if relative time selected • in the range 0 to 23, if absolute time selected

- `u16FinishTime` is a bitmap representing the stop-time of a 'running' cycle of the appliance, as follows:

Bits	Description
0-5	Minutes part of the stop-time, in the range 0 to 59 (may be absolute or relative time - see below)
6-7	Type of time encoding: • 0x0: Relative time - stop-time is a delay from the time that the attribute was set • 0x1: Absolute time - stop-time is an actual time of the 24-hour clock • 0x2-0x3: Reserved The defaults are absolute time for ovens and relative time for other appliances.
8-15	Hours part of the stop-time: • in the range 0 to 255, if relative time selected • in the range 0 to 23, if absolute time selected

- `u16RemainingTime` is an optional attribute indicating the time, in minutes, remaining in the current 'running' cycle of the appliance (time until the end of the cycle) - this attribute is constantly updated during the running cycle and is zero when the appliance is not running
- `u8AttributeReportingStatus` is an optional attribute that should be enabled when attribute reporting is used for the cluster (see Section 2.3.5). The value of this attribute indicates whether there are attribute reports still pending (0x00) or the attribute reports are complete (0x01) - all other values are reserved. This attribute is also described in Section 2.4.
- `u16ClusterRevision` is a mandatory attribute that specifies the revision of the cluster specification on which this cluster instance is based. The cluster specification in the ZCL r6 corresponds to a cluster revision of 1. The value is incremented by one for each subsequent revision of the cluster specification. This attribute is also described in Section 2.4.

45.3 Attributes for default reporting

The following attributes of the Appliance Control cluster can be selected for default reporting:

```
u16StartTime
u16FinishTime
```

- `u16RemainingTime`

Attribute reporting (including default reporting) is described in Appendix B. Enabling reports for these attributes is described in Appendix B.3.6.

45.4 Sending commands

The Appliance Control cluster server resides on the appliance to be controlled (e.g. a washing machine) and the cluster client resides on the controlling device (normally a remote control unit).

The commands from the client to the server can be of two types:

- 'Execution' commands, requesting appliance operations
- 'Status' commands, requesting appliance status information

In addition, status notification messages can be sent unsolicited from the server to the client.

Sending the above messages is described in the sub-sections below.

45.4.1 Execution Commands from Client to Server

An ‘execution’ command can be sent from the client to request that an operation is performed on the appliance (server) - the request is sent in an ‘Execution of Command’ message. The application on the client can send this message by calling the function **eCLD_ACExecutionOfCommandSend()**.

The possible operations depend on the target appliance but the following operations are available to be specified in the message payload (described in Section 45.9.2):

- Start appliance cycle
- Stop appliance cycle
- Pause appliance cycle
- Start superfreezing cycle
- Stop superfreezing cycle
- Start supercooling cycle
- Stop supercooling cycle
- Disable gas
- Enable gas

In the start and stop commands, the start-time and end-time can be specified. The commands are fully detailed in the British Standards document BS EN 50523.

The application on the server (appliance) will be notified of the received command by an E_CLD_APPLIANCE_CONTROL_CMD_EXECUTION_OF_COMMAND event (Appliance Control events are described in Section 45.5). The required command is specified in the payload of the message, which is contained in the above event. The application must then perform the requested command (if possible).

45.4.2 Status Commands from Client to Server

The application on the cluster client can request the current status of the appliance by sending a ‘Signal State’ message to the cluster server on the appliance. This message can be sent by calling the function **eCLD_ACSignalStateSend()**. This function returns immediately and the requested status information is later returned in an E_CLD_APPLIANCE_CONTROL_CMD_SIGNAL_STATE_RESPONSE event, which is generated when a response arrives from the server (Appliance Control events are described in Section 45.5).

Note: *The cluster server handles the ‘Signal State’ message automatically and returns the requested status information in a ‘Signal State Response’ message to the client.*

The appliance status information from the message payload is contained in the above event - for details of this payload and the status information, refer to Section 45.9.3.

45.4.3 Status Notifications from Server to Client

The cluster server on the appliance can send unsolicited status notifications to the client in ‘Signal State Notification’ messages. A message of this kind can be sent by the application on the server by calling either of the following functions:

- **eCLD_ACSignalStateNotificationSend()**
- **eCLD_ACSignalStateResponseORSignalStateNotificationSend()**

Note: *The latter function is also used internally by the cluster server to send a ‘Signal State Response’ message - see Section 45.4.2.*

The appliance status information from the ‘Signal State Notification’ message is reported to the application on the cluster client through the event E_CLD_APPLIANCE_CONTROL_CMD_SIGNAL_STATE_NOTIFICATION, which is generated when the notification arrives from the server (Appliance Control events are described in

Section 45.5). The appliance status information from the message payload is contained in the above event - for details of this payload and the status information, refer to Section 45.9.3.

45.5 Appliance control events

The Appliance Control cluster has its own events that are handled through the callback mechanism described in Chapter 3. The cluster contains its own event handler. If a device uses this cluster then application-specific Appliance Control event handling must be included in the user-defined callback function for the associated endpoint, where this callback function is registered through the relevant endpoint registration function. This callback function will then be invoked when an Appliance Control event occurs and needs the attention of the application.

For an Appliance Control 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 `tsCLD_ApplianceControlCallBackMessage` structure:

```
typedef struct
{
    uint8   u8CommandId;
    bool   *pbApplianceStatusTwoPresent;
    union
    {
        tsCLD_AC_ExecutionOfCommandPayload *psExecutionOfCommandPayload;
        tsCLD_AC_SignalStateResponseORSignalStateNotificationPayload

 *psSignalStateResponseAndNotificationPayload;
    } uMessage;
} tsCLD_ApplianceControlCallBackMessage;
```

When an Appliance Control event occurs, one of four 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 are detailed the tables below for events generated on a server and a client.

Table 106. Appliance Control Command Types (Events on Server)

u8CommandId Enumeration	Description
E_CLD_APPLIANCE_CONTROL_ CMD_EXECUTION_OF_COMMAND	An ‘Execution of Command’ message has been received by the server (appliance), requesting an opera-tion on the appliance
E_CLD_APPLIANCE_CONTROL_ CMD_SIGNAL_STATE	A ‘Signal State’ message has been received by the server (appliance), requesting the status of the appli-ance

Table 107. Appliance Control Command Types (Events on Client)

u8CommandId Enumeration	Description
E_CLD_APPLIANCE_CONTROL_CM-D_SIGNAL_ STATE_RESPONSE	A response to a ‘Signal State’ message has been received by the client, containing the requested appli-ance status
E_CLD_APPLIANCE_CONTROL_CM-D_SIGNAL_ STATE_NOTIFICATION	A ‘Signal State’ notification message has been received by the client, containing unsolicited status information

45.6 Functions

The following Appliance Control cluster functions are provided:

1. eCLD_ApplianceControlCreateApplianceControl
2. eCLD_ACExecutionOfCommandSend
3. eCLD_ACSignalStateSend
4. eCLD_ACSignalStateResponseORSignalStateNotificationSend
5. eCLD_ACSignalStateNotificationSend
6. eCLD_ACChangeAttributeTime

45.6.1 eCLD_ApplianceControlCreateApplianceControl

```
teZCL_Status eCLD_ApplianceControlCreateApplianceControl(
        tsZCL_ClusterInstance *psClusterInstance,
        bool_t bIsServer,
        tsZCL_ClusterDefinition *psClusterDefinition,
        void *pvEndPointSharedStructPtr,
        uint8 *pu8AttributeControlBits,
        tsCLD_ApplianceControlCustomDataStructure
         *psCustomDataStructure);
```

Description

This function creates an instance of the Appliance Control 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 an Appliance Control 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 D.

Note: *This function must not be called for an endpoint on which a standard ZigBee device is 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 the ZigBee Devices User Guide (JNUG3131).*

When used, this function must be the first Appliance Control cluster function called in the application, and must be called after the stack has been started and after the ZCL has been initialized.

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 is automatically adjusted by the compiler using the following declaration:

```
uint8 au8ApplianceControlAttributeControlBits
[(sizeof(asCLD_ApplianceControlClusterAttributeDefinitions) /
 sizeof(tsZCL_AttributeDefinition))];
```

Parameters

- *psClusterInstance* Pointer to structure containing information about the cluster instance to be created (see Section 6.1.16). This structure is updated by the function by initialising individual structure fields.
- *bIsServer* Type of cluster instance (server or client) to be created:
  - TRUE - server
  - FALSE - client

- *psClusterDefinition* Pointer to structure indicating the type of cluster to be created (see Section 6.1.2). In this case, this structure must contain the details of the Appliance Control cluster. This parameter can refer to a pre-filled structure called `sCLD_ApplianceControl` which is provided in the **ApplianceControl.h** file.
- *pvEndPointSharedStructPtr* Pointer to the shared structure used for attribute storage. This parameter should be the address of the structure of type `tsCLD_ApplianceControl` which defines the attributes of Appliance Control cluster. The function initializes the attributes with default values.
- *pu8AttributeControlBits* Pointer to an array of **uint8** values, with one element for each attribute in the cluster (see above). For a cluster client, set this pointer to NULL.
- *psCustomDataStructure* Pointer to a structure containing the storage for internal functions of the cluster (see Section 45.9.4)

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_INVALID_VALUE

45.6.2 eCLD_ACExecutionOfCommandSend

```
teZCL_Status eCLD_ACExecutionOfCommandSend(
        uint8 u8SourceEndPointId,
        uint8 u8DestinationEndPointId,
        tsZCL_Address *psDestinationAddress,
        uint8 *pu8TransactionSequenceNumber,
        tsCLD_AC_ExecutionOfCommandPayload *psPayload);
```

Description

This function can be used on an Appliance Control cluster client to send an ‘Execution of Command’ message to a cluster server (appliance), where this message may specify one of the following control commands:

- Start appliance cycle
- Stop appliance cycle
- Pause appliance cycle
- Start superfreezing cycle
- Stop superfreezing cycle
- Start supercooling cycle
- Stop supercooling cycle
- Disable gas
- Enable gas

The required command is specified in the payload of the message (a pointer to this payload must be provided). The commands are fully detailed in the British Standards document BS EN 50523.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the message. The TSN in the response is 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.

Parameter

- *u8SourceEndPointId* Number of the local endpoint through which to send the message. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values
- *u8DestinationEndPointId* Number of the endpoint on the remote node to which the message is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP
- *psDestinationAddress* Pointer to a structure holding the address of the node to which the request is sent
- *pu8TransactionSequenceNumber* Pointer to a location to receive the Transaction Sequence Number (TSN) of the message
- *psPayload* Pointer to a structure containing the payload for the message (see Section 45.9.2).

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_INVALID_VALUE

45.6.3 eCLD_ACSignalStateSend

```
teZCL_Status eCLD_ACSignalStateSend(
        uint8 u8SourceEndPointId,
        uint8 u8DestinationEndPointId,
        tsZCL_Address *psDestinationAddress,
        uint8 *pu8TransactionSequenceNumber);
```

Description

This function can be used on an Appliance Control cluster client to send a ‘Signal State’ message to a cluster server (appliance), which requests the status of the appliance. The function returns immediately and the requested status information is later returned in the following event, which is generated when a response is received from the server:

`E_CLD_APPLIANCE_CONTROL_CMD_SIGNAL_STATE_RESPONSE`

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the message. The TSN in the response is 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 to send the message. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values
- *u8DestinationEndPointId*: Number of the endpoint on the remote node to which the message is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP
- *psDestinationAddress*: Pointer to a structure holding the address of the node to which the message is sent
- *pu8TransactionSequenceNumber*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the message

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_INVALID_VALUE

45.6.4 eCLD_ACSignalStateResponseORSignalStateNotificationSend

```
teZCL_Status eCLD_ACSignalStateResponseORSignalStateNotificationSend(
        uint8 u8SourceEndPointId,
        uint8 u8DestinationEndPointId,
        tsZCL_Address *psDestinationAddress,
        uint8 *pu8TransactionSequenceNumber,
        teCLD_ApplianceControl_ServerCommandId eCommandId,
        bool bApplianceStatusTwoPresent,
        tsCLD_AC_SignalStateResponseORSignalStateNotificationPayload
 *psPayload);
```

Description

This function can be used on an Appliance Control cluster server to send a ‘Signal State Response’ message (in reply to a ‘Signal State Request’ message) or an unsolicited ‘Signal State Notification’ message to a cluster client.

The command to be sent must be specified as one of:

- E_CLD_APPLIANCE_CONTROL_CMD_SIGNAL_STATE_RESPONSE
- E_CLD_APPLIANCE_CONTROL_CMD_SIGNAL_STATE_NOTIFICATION

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the message. The TSN in the response is 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 to send the message. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values
- *u8DestinationEndPointId* Number of the endpoint on the remote node to which the message is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP
- *psDestinationAddress* Pointer to a structure holding the address of the node to which the message is sent
- *pu8TransactionSequenceNumber* Pointer to a location to receive the Transaction Sequence Number (TSN) of the message
- *eCommandId* Enumeration indicating the command to be sent (see above and Section 45.8.3)
- *bApplianceStatusTwoPresent* Boolean indicating whether additional appliance status data is present in payload:
  - TRUE - Present
  - FALSE - Not present
- *psPayload* Pointer to structure containing payload for message (see above and Section 45.9.3)

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_INVALID_VALUE

45.6.5 eCLD_ACSignalStateNotificationSend

```
teZCL_Status eCLD_ACSignalStateNotificationSend(
        uint8 u8SourceEndPointId,
        uint8 u8DestinationEndPointId,
        tsZCL_Address *psDestinationAddress,
        uint8 *pu8TransactionSequenceNumber,
        bool bApplianceStatusTwoPresent,
        tsCLD_AC_SignalStateResponseORSignalStateNotificationPayload
 *psPayload);
```

Description

This function can be used on an Appliance Control cluster server to send an unsolicited ‘Signal State Notification’ message to a cluster client. The function is an alternative to **eCLD_ACSignalStateResponse ORSignalStateNotificationSend()**.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the message. The TSN in the response is 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 to send the message. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values
- *u8DestinationEndPointId*: Number of the endpoint on the remote node to which the message is sent. This parameter is ignored when sending to address types `eZCL_AMBOUND` and `eZCL_AMGROUP`
- *psDestinationAddress*: Pointer to a structure holding the address of the node to which the message is sent
- *pu8TransactionSequenceNumber*: Pointer to a location to receive the Transaction Sequence Number (TSN) of the message
- *bApplianceStatusTwoPresent*: Boolean indicating whether additional appliance status data is present in payload:
  - TRUE - Present
  - FALSE - Not present
- *psPayload*: Pointer to structure containing payload for message (see above and Section 45.9.3)

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_INVALID_VALUE

45.6.6 eCLD_ACChangeAttributeTime

```
teZCL_Status eCLD_ACChangeAttributeTime(
        uint8 u8SourceEndPointId,
        teCLD_ApplianceControl_Cluster_AttrID eAttributeTimeId,
        uint16 u16TimeValue);
```

Description

This function can be used on an Appliance Control cluster server (appliance) to update the time attributes of the cluster (start time, finish time, remaining time). This is particularly useful if the host node has its own timer.

The target attribute must be specified using one of:

- E_CLD_APPLIANCE_CONTROL_ATTR_ID_START_TIME
- E_CLD_APPLIANCE_CONTROL_ATTR_ID_FINISH_TIME
- E_CLD_APPLIANCE_CONTROL_ATTR_ID_REMAINING_TIME

Parameters

- *u8SourceEndPointId*: Number of the local endpoint through which to send the message. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values
- *eAttributeTimeId*: Identifier of attribute to be updated (see above and Section 45.9.1)
- *u16TimeValue*: UTC time to set

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_INVALID_VALUE

45.7 Return codes

The Appliance Control cluster functions use the ZCL return codes, listed in Section 7.2.

45.8 Enumerations

45.8.1 'Attribute ID' Enumerations

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

```
typedef enum PACK
{
    E_CLD_APPLIANCE_CONTROL_ATTR_ID_START_TIME     = 0x0000,
    E_CLD_APPLIANCE_CONTROL_ATTR_ID_FINISH_TIME,
    E_CLD_APPLIANCE_CONTROL_ATTR_ID_REMAINING_TIME
} teCLD_ApplianceControl_Cluster_AttrID;
```

45.8.2 'Client Command ID' Enumerations

The following enumerations are used in commands issued on a cluster client.

```
typedef enum PACK
{
    E_CLD_APPLIANCE_CONTROL_CMD_EXECUTION_OF_COMMAND = 0x00,
    E_CLD_APPLIANCE_CONTROL_CMD_SIGNAL_STATE,
} teCLD_ApplianceControl_ClientCommandId;
```

The above enumerations are described in the table below.

Table 108. 'Client Command ID' Enumerations

Enumeration	Description
E_CLD_APPLIANCE_CONTROL_CMD_EXECUTION_OF_COMMAND	'Execution of Command' message
E_CLD_APPLIANCE_CONTROL_CMD_SIGNAL_STATE	'Signal State' message

45.8.3 'Server command ID' enumerations

The following enumerations are used in commands issued on a cluster server.

```
typedef enum PACK
{
    E_CLD_APPLIANCE_CONTROL_CMD_SIGNAL_STATE_RESPONSE = 0x00,
    E_CLD_APPLIANCE_CONTROL_CMD_SIGNAL_STATE_NOTIFICATION
} teCLD_ApplianceControl_ServerCommandId;
```

The above enumerations are described in the table below.

Table 109. 'Server command ID' enumerations

Enumeration	Description
E_CLD_APPLIANCE_CONTROL_CMD_SIGNAL_STATE_ RESPONSE	A response to a 'Signal State' request
E_CLD_APPLIANCE_CONTROL_CMD_SIGNAL_STATE_ NOTIFICATION	A 'Signal State' notification

45.9 Structures

45.9.1 tsCLD_ApplianceControlCallBackMessage

For an Appliance Control 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 `tsCLD_ApplianceControlCallBackMessage` structure:

```
typedef struct
{
    uint8    u8CommandId;
    bool    *pbApplianceStatusTwoPresent;
    union
    {
        tsCLD_AC_ExecutionOfCommandPayload
 *psExecutionOfCommandPayload;
```

```
        tsCLD_AC_SignalStateResponseORSignalStateNotificationPayload


 *psSignalStateResponseAndNotificationPayload;
    } uMessage;
} tsCLD_ApplianceControlCallBackMessage;
```

where:

- `u8CommandId` indicates the type of Appliance Control command that has been received, one of:
  - E_CLD_APPLIANCE_CONTROL_CMD_EXECUTION_OF_COMMAND
  - E_CLD_APPLIANCE_CONTROL_CMD_SIGNAL_STATE
  - E_CLD_APPLIANCE_CONTROL_CMD_SIGNAL_STATE_RESPONSE
  - E_CLD_APPLIANCE_CONTROL_CMD_SIGNAL_STATE_NOTIFICATION
- `pbApplianceStatusTwoPresent` is a pointer to a boolean indicating whether a second set of non-standard or proprietary status data is available:
  - TRUE - additional status data available
  - FALSE - additional status data unavailable
- `uMessage` is a union containing the command payload as one of (depending on the value of `u8CommandId`):
- `psExecutionOfCommandPayload` is a pointer to the payload of an ‘Execution of Command’ message (see Section 45.9.2)
- `psSignalStateResponseAndNotificationPayload` is a pointer to the payload of a ‘Signal State’ response or notification message (see Section 45.9.3)

45.9.2 tsCLD_AC_ExecutionOfCommandPayload

This structure contains the payload for an "Execution of Command" message.

```
typedef struct
{
            zenum8                  eExecutionCommandId;
} stsCLD_AC_ExecutionOfCommandPayload;
```

where `eExecutionCommandId` is a value representing the command to be executed - the commands are detailed in the British Standards document BS EN 50523.

45.9.3 tsCLD_AC_SignalStateResponseORSignalStateNotificationPayload

This structure contains the payload for a "Signal State" response or notification message.

```
typedef struct
{
           zenum8   eApplianceStatus;
           zuint8                  u8RemoteEnableFlagAndDeviceStatus;
           zuint24                         u24ApplianceStatusTwo;
} tsCLD_AC_SignalStateResponseORSignalStateNotificationPayload;
```

where:

- `eApplianceStatus` is a value indicating the reported appliance status (the relevant status values depend on the appliance):

Status Value	Description
0x00	Reserved
0x01	Appliance in off state

Status Value	Description
0x02	Appliance in stand-by
0x03	Appliance already programmed
0x04	Appliance already programmed and ready to start
0x05	Appliance is running
0x06	Appliance is in pause state
0x07	Appliance end programmed tasks
0x08	Appliance is in a failure state
0x09	Appliance programmed tasks have been interrupted
0x0A	Appliance in idle state
0x0B	Appliance rinse hold
0x0C	Appliance in service state
0x0D	Appliance in superfreezing state
0x0E	Appliance in supercooling state
0x0F	Appliance in superheating state
0x10-0x3F	Reserved
0x40-0x7F	Non-standardised
0x80-0xFF	Proprietary

- `u8RemoteEnableFlagAndDeviceStatus` is a bitmap value indicating the status of the relationship between the appliance and the remote control unit as well as the type of additional status information reported in `u24ApplianceStatusTwo`:

Bits	Field	Values/Description
0-3	Remote Enable Flags	Status of remote control link: • 0x0: Disabled • 0x1: Enabled remote and energy control • 0x2-0x06: Reserved • 0x7: Temporarily locked/disabled • 0x8-0xE: Reserved • 0xF: Enabled remote control
4-7	Device Status 2	Type of information in `u24ApplianceStatusTwo`: • 0x0: Proprietary • 0x1: Proprietary • 0x2: IRIS symptom code • 0x3-0xF: Reserved

- `u24ApplianceStatusTwo` is a value indicating non-standard or proprietary status information about the appliance. The type of status information represented by this value is indicated in the ‘Device Status 2’ field of `u8RemoteEnableFlagAndDeviceStatus`. In the case of an IRIS symptom code, the three bytes of this value represent a 3-digit code.

45.9.4 tsCLD_ApplianceControlCustomDataStructure

The Appliance Control cluster requires extra storage space to be allocated to be used by internal functions. The structure definition for this storage is shown below:

```
typedef struct
{
    tsZCL_ReceiveEventAddress                sReceiveEventAddress;
    tsZCL_CallBackEvent                      sCustomCallBackEvent;
    tsCLD_ApplianceControlCallBackMessage    sCallBackMessage;
} tsCLD_ApplianceControlCustomDataStructure;
```

The fields are for internal use and no knowledge of them is required.

45.10 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 Appliance Control cluster.

To enable the Appliance Control cluster in the code to be built, it is necessary to add the following line to the file:

```
#define CLD_APPLIANCE_CONTROL
```

In addition, to enable the cluster as a client or server, it is also necessary to add one of the following lines to the same file:

```
#define APPLIANCE_CONTROL_SERVER
#define APPLIANCE_CONTROL_CLIENT
```

The Appliance Control cluster contains macros that may be optionally specified at compile-time by adding one or more of the following lines to the **zcl_options.h** file.

Optional attributes

Add this line to enable the optional Time Remaining attribute:

```
#define CLD_APPLIANCE_CONTROL_REMAINING_TIME
```

Global Attributes

Add this line to enable the optional Attribute Reporting Status attribute:

```
#define CLD_LEVELCONTROL_ATTR_ID_ATTRIBUTE_REPORTING_STATUS
```

Add this line to define the value (n) of the Cluster Revision attribute:

```
#define CLD_LEVELCONTROL_CLUSTER_REVISION <n>
```

The default value is 1, which corresponds to the revision of the cluster in the ZCL r6 specification (see Section 2.4).

46 Appliance Identification Cluster

This chapter outlines the Appliance Identification cluster, which provides an interface for obtaining and setting basic appliance information.

The Appliance Identification cluster has a Cluster ID of 0x0B00.

46.1 Overview

The Appliance Identification cluster provides an interface for obtaining and setting information about an appliance, such as product type and manufacturer.

The cluster is enabled by defining CLD_APPLIANCE_IDENTIFICATION in the **zcl_options.h** file. Further compile-time options for the Appliance Identification cluster are detailed in Section 46.6.

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

- Basic Appliance Identification
- Extended Appliance Identification

46.2 Cluster structure and attributes

The structure definition for the Appliance Identification cluster (server) is:

```
typedef struct
{
#ifdef APPLIANCE_IDENTIFICATION_SERVER
    zbmap56   u64BasicIdentification;
#ifdef CLD_APPLIANCE_IDENTIFICATION_ATTR_COMPANY_NAME
tsZCL_CharacterString   sCompanyName;
uint8   au8CompanyName[16];
#endif
#ifdef CLD_APPLIANCE_IDENTIFICATION_ATTR_COMPANY_ID
    zuint16  u16CompanyId;
#endif
#ifdef CLD_APPLIANCE_IDENTIFICATION_ATTR_BRAND_NAME
    tsZCL_CharacterString  sBrandName;
    uint8    au8BrandName[16];
#endif
#ifdef CLD_APPLIANCE_IDENTIFICATION_ATTR_BRAND_ID
    zuint16   u16BrandId;
#endif
#ifdef CLD_APPLIANCE_IDENTIFICATION_ATTR_MODEL
    tsZCL_OctetStrings   Model;
    uint8 au8Model[16];
#endif
#ifdef CLD_APPLIANCE_IDENTIFICATION_ATTR_PART_NUMBER
    tsZCL_OctetString  sPartNumber;
    uint8   au8PartNumber[16];
#endif
#ifdef CLD_APPLIANCE_IDENTIFICATION_ATTR_PRODUCT_REVISION
    tsZCL_OctetString    sProductRevision;
    uint8     au8ProductRevision[6];
#endif
#ifdef CLD_APPLIANCE_IDENTIFICATION_ATTR_SOFTWARE_REVISION
    tsZCL_OctetString    sSoftwareRevision;
    uint8     au8SoftwareRevision[6];
#endif
#ifdef CLD_APPLIANCE_IDENTIFICATION_ATTR_PRODUCT_TYPE_NAME
    tsZCL_OctetString    sProductTypeName;
    uint8     au8ProductTypeName[2];
#endif
```

```
#ifdef CLD_APPLIANCE_IDENTIFICATION_ATTR_PRODUCT_TYPE_ID
    zuint16   u16ProductTypeId;
#endif
#ifdef CLD_APPLIANCE_IDENTIFICATION_ATTR_CECED_SPEC_VERSION
    zuint8   u8CECEDSpecificationVersion;
#endif
#endif
    zuint16   u16ClusterRevision;
} tsCLD_ApplianceIdentification;
```

where:

‘Basic Appliance Identification’ Attribute Set

- `u64BasicIdentification` is a mandatory attribute which is a 56-bit bitmap containing the following information about the appliance:

Bits	Information
0-15	Company (manufacturer) ID
16-31	Brand ID
32-47	Product Type ID, one of: • 0x0000: White Goods • 0x5601: Dishwasher • 0x5602: Tumble Dryer • 0x5603: Washer Dryer • 0x5604: Washing Machine • 0x5E03: Hob • 0x5E09: Induction Hob • 0x5E01: Oven • 0x5E06: Electrical Oven • 0x6601: Refrigerator/Freezer For enumerations, see Section 46.5.2.
48-55	Specification Version

‘Extended Appliance Identification’ Attribute Set

- The following optional pair of attributes are used to store human readable versions of the company (manufacturer) name:
  - `sCompanyName` is a `tsZCL_OctetString` structure which contains a character string representing the company name of up to 16 characters
  - `au8CompanyName[16]` is a byte-array which contains the character data bytes representing the company name
- `u16CompanyId` is an optional attribute which contains the company ID
- The following optional pair of attributes are used to store human readable versions of the brand name:
  - `sBrandName` is a `tsZCL_OctetString` structure which contains a character string representing the brand name of up to 16 characters
  - `au8BrandName[16]` is a byte-array which contains the character data bytes representing the brand name
- `u16BrandId` is an optional attribute which contains the brand ID
- The following optional pair of attributes are used to store human readable versions of the manufacturer-defined model name:
  - `sModel` is a `tsZCL_OctetString` structure which contains a character string representing the model name of up to 16 characters

  - au8Model[16] is a byte-array which contains the character data bytes representing the model name
- The following optional pair of attributes are used to store human readable versions of the manufacturer-defined part number/code:
  - sPartNumber is a tsZCL_OctetString structure which contains a character string representing the part number/code of up to 16 characters
  - au8PartNumber[16] is a byte-array which contains the character data bytes representing the part number/code
- The following optional pair of attributes are used to store human readable versions of the manufacturer-defined product revision number:
  - sProductRevision is a tsZCL_OctetString structure which contains a character string representing the product revision number of up to 6 characters
  - au8ProductRevision[6] is a byte-array which contains the character data bytes representing the product revision number
- The following optional pair of attributes are used to store human readable versions of the manufacturer-defined software revision number:
  - sSoftwareRevision is a tsZCL_OctetString structure which contains a character string representing the software revision number of up to 6 characters
  - au8SoftwareRevision[6] is a byte-array which contains the character data bytes representing the software revision number
- The following optional pair of attributes are used to store human readable versions of the 2-character product type name (e.g. "WM" for washing machine):
  - sProductTypeName is a tsZCL_OctetString structure which contains a character string representing the product type name of up to 2 characters
  - au8ProductTypeName[2] is a byte-array which contains the character data bytes representing the product type name
- u16ProductTypeId is an optional attribute containing the product type ID (from those listed above in the description of u64BasicIdentification)
- u8CECEDSpecificationVersion is an optional attribute which indicates the version of the CECED specification to which the appliance conforms, from the following:

Value	Specification
0x10	Compliant with v1.0, not certified
0x1A	Compliant with v1.0, certified
0xX0	Compliant with vX.0, not certified
0xXA	Compliant with vX.0, certified

Global Attributes

- u16ClusterRevision is a mandatory attribute that specifies the revision of the cluster specification on which this cluster instance is based. The cluster specification in the ZCL r6 corresponds to a cluster revision of 1. The value is incremented by one for each subsequent revision of the cluster specification. This attribute is also described in Section 2.4.

46.3 Functions

The following Appliance Identification cluster function is provided:

eCLD_ApplianceIdentificationCreateApplianceIdentification

Note: *The attributes of this cluster can be accessed using the attribute read/write functions detailed in Section 5.2.*

46.3.1 eCLD_ApplianceIdentificationCreateAppliance Identification

```
teZCL_Status eCLD_ApplianceIdentificationCreateApplianceIdentification(
   tsZCL_ClusterInstance *psClusterInstance,
   bool_t bIsServer,
   tsZCL_ClusterDefinition *psClusterDefinition,
   void *pvEndPointSharedStructPtr,
   uint8 *pu8AttributeControlBits);
```

Description

This function creates an instance of the Appliance Identification 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 an Appliance Identification 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 D.

Note: *This function must not be called for an endpoint on which a standard ZigBee device is 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 the ZigBee Devices User Guide (JNUG3131).*

When used, this function must be the first Appliance Identification cluster function called in the application, and must be called after the stack has been started and after the ZCL has been initialized.

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 is automatically adjusted by the compiler using the following declaration:

```
uint8 au8ApplianceIdentificationAttributeControlBits
[(sizeof(asCLD_ApplianceIdentificationClusterAttributeDefinitions)
        / sizeof(tsZCL_AttributeDefinition))];
```

Parameters

- *psClusterInstance*: Pointer to structure containing information about the cluster instance to be created (see Section 6.1.16). This structure is updated by the function by initializing individual structure fields.
- *bIsServer*: Type of cluster instance (server or client) to be created:
- TRUE - server
- FALSE - client
- *psClusterDefinition*: Pointer to structure indicating the type of cluster to be created (see Section 6.1.2). In this case, this structure must contain the details of the Appliance Identification cluster. This parameter can refer to a pre-filled structure called `sCLD_ApplianceIdentification` which is provided in the **ApplianceIdentification.h** file.
- *pvEndPointSharedStructPtr*: Pointer to the shared structure used for attribute storage. This parameter should be the address of the structure of the `tsCLD_ApplianceIdentification` type which defines the attributes of Appliance Identification cluster. The function initializes the attributes with default values.

- *pu8AttributeControlBits*: Pointer to an array of **uint8** values, with one element for each attribute in the cluster (see above). For a cluster client, set this pointer to NULL.

Returns

E_ZCL_SUCCESS
E_ZCL_FAIL
E_ZCL_ERR_PARAMETER_NULL
E_ZCL_ERR_INVALID_VALUE

46.4 Return codes

The Appliance Identification cluster function uses the ZCL return codes, listed in Section 7.2.

46.5 Enumerations

46.5.1 'Attribute ID' enumerations

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

```
typedef enum PACK
{
    E_CLD_APPLIANCE_IDENTIFICATION_ATTR_ID_BASIC_IDENTIFICATION = 0x0000,
    E_CLD_APPLIANCE_IDENTIFICATION_ATTR_ID_COMPANY_NAME = 0x0010,
    E_CLD_APPLIANCE_IDENTIFICATION_ATTR_ID_COMPANY_ID,
    E_CLD_APPLIANCE_IDENTIFICATION_ATTR_ID_BRAND_NAME,
    E_CLD_APPLIANCE_IDENTIFICATION_ATTR_ID_BRAND_ID,
    E_CLD_APPLIANCE_IDENTIFICATION_ATTR_ID_MODEL,
    E_CLD_APPLIANCE_IDENTIFICATION_ATTR_ID_PART_NUMBER,
     E_CLD_APPLIANCE_IDENTIFICATION_ATTR_ID_PRODUCT_REVISION,
    E_CLD_APPLIANCE_IDENTIFICATION_ATTR_ID_SOFTWARE_REVISION,
    E_CLD_APPLIANCE_IDENTIFICATION_ATTR_ID_PRODUCT_TYPE_NAME,
    E_CLD_APPLIANCE_IDENTIFICATION_ATTR_ID_PRODUCT_TYPE_ID,
    E_CLD_APPLIANCE_IDENTIFICATION_ATTR_ID_CECED_SPEC_VERSION
} teCLD_ApplianceIdentification_Cluster_AttrID;
```

46.5.2 'Product Type ID' enumerations

The following enumerations are used to represent the set of product type IDs.

```
typedef enum PACK
{
 E_CLD_AI_PT_ID_WHITE_GOODS    = 0x0000,
  E_CLD_AI_PT_ID_DISHWASHER  = 0x5601,
  E_CLD_AI_PT_ID_TUMBLE_DRYER,
  E_CLD_AI_PT_ID_WASHER_DRYER,
  E_CLD_AI_PT_ID_WASHING_MACHINE,
  E_CLD_AI_PT_ID_HOBS            = 0x5E03,
  E_CLD_AI_PT_ID_INDUCTION_HOBS    = 0x5E09,
  E_CLD_AI_PT_ID_OVEN         = 0x5E01,
  E_CLD_AI_PT_ID_ELECTRICAL_OVEN   = 0x5E06,
  E_CLD_AI_PT_ID_REFRIGERATOR_FREEZER  = 0x6601
} teCLD_ApplianceIdentification_ProductTypeId;
```

46.6 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 Appliance Identification cluster.

To enable the Appliance Identification cluster in the code to be built, it is necessary to add the following line to the file:

```
#define CLD_APPLIANCE_IDENTIFICATION
```

In addition, to enable the cluster as a client or server, it is also necessary to add one of the following lines to the same file:

```
#define APPLIANCE_IDENTIFICATION_SERVER
#define APPLIANCE_IDENTIFICATION_CLIENT
```

The Appliance Identification cluster contains macros that may be optionally specified at compile-time by adding one or more of the following lines to the **zcl_options.h** file.

Optional Attributes

Add this line to enable the optional Company Name attributes:

```
#define CLD_APPLIANCE_IDENTIFICATION_ATTR_COMPANY_NAME
```

Add this line to enable the optional Company ID attributes:

```
#define CLD_APPLIANCE_IDENTIFICATION_ATTR_COMPANY_ID
```

Add this line to enable the optional Brand Name attributes:

```
#define CLD_APPLIANCE_IDENTIFICATION_ATTR_BRAND_NAME
```

Add this line to enable the optional Brand ID attribute:

```
#define CLD_APPLIANCE_IDENTIFICATION_ATTR_BRAND_ID
```

Add this line to enable the optional Model attributes:

```
#define CLD_APPLIANCE_IDENTIFICATION_ATTR_MODEL
```

Add this line to enable the optional Part Number attributes:

```
#define CLD_APPLIANCE_IDENTIFICATION_ATTR_PART_NUMBER
```

Add this line to enable the optional Product Revision attributes:

```
#define CLD_APPLIANCE_IDENTIFICATION_ATTR_PRODUCT_REVISION
```

Add this line to enable the optional Software Revision attributes:

```
#define CLD_APPLIANCE_IDENTIFICATION_ATTR_SOFTWARE_REVISION
```

Add this line to enable the optional Product Type Name attributes:

```
#define CLD_APPLIANCE_IDENTIFICATION_ATTR_PRODUCT_TYPE_NAME
```

Add this line to enable the optional Product Type ID attribute:

```
#define CLD_APPLIANCE_IDENTIFICATION_ATTR_PRODUCT_TYPE_ID
```

Add this line to enable the optional CECED Specification Version attribute:

```
#define CLD_APPLIANCE_IDENTIFICATION_ATTR_CECED_SPEC_VERSION
```

Global Attributes

Add this line to define the value (n) of the Cluster Revision attribute:

```
#define CLD_APPLIANCE_IDENTIFICATION_CLUSTER_REVISION <n>
```

The default value is 1, which corresponds to the revision of the cluster in the ZCL r6 specification (see Section 2.4).

47 Appliance Events and Alerts Cluster

This chapter outlines the Appliance Events and Alerts cluster, which provides an interface for the notification of significant events and alert situations.

The Appliance Events and Alerts cluster has a Cluster ID of 0x0B02.

47.1 Overview

The Appliance Events and Alerts cluster provides an interface for sending notifications of appliance events (for example, target temperature reached) and alerts (for example, alarms).

The cluster is enabled by defining CLD_APPLIANCE_EVENTS_AND_ALERTS in the **zcl_options.h** file. Further compile-time options for the Appliance Events and Alerts cluster are detailed in Section 47.9.

Events are notified in terms of header and event identifier fields (an event may occur when the appliance reaches a certain state, such as the end of its operational cycle).

Alerts are notified in terms of the following fields:

- Alert identification value
- Alert category, one of: Warning, Danger, Failure
- Presence/recovery flag (indicating alert has been either detected or recovered)

47.2 Cluster structure and attribute

The structure definition for the Appliance Events and Alerts cluster (server) is.

```
typedef struct
{
 zuint16   u16ClusterRevision;
} tsCLD_ApplianceEventsAndAlerts;
```

where `u16ClusterRevision` is a mandatory attribute that specifies the revision of the cluster specification on which this cluster instance is based. The cluster specification in the ZCL r6 corresponds to a cluster revision of 1. The value is incremented by one for each subsequent revision of the cluster specification. This attribute is also described in Section 2.4.

47.3 Sending Messages

The Appliance Events and Alerts cluster server resides on the appliance (for example, a washing machine) and the cluster client resides on a controlling device (normally a remote control unit).

Messages can be sent between the client and the server in the following ways:

- Alerts that are active on the appliance can be requested by the client by sending a ‘Get Alerts’ message to the server (which replies with a ‘Get Alerts Response’ message).
- Alerts that are active on the appliance can be sent unsolicited from the server to the client in an ‘Alerts Notification’ message.
- The server can notify the client of an appliance event by sending an unsolicited ‘Event Notification’ message to the client

Sending the above messages is described in the sub-sections below.

47.3.1 'Get Alerts' Messages from Client to Server

The application on the cluster client can request the alerts that are currently active on the appliance by sending a 'Get Alerts' message to the server - this message is sent by calling the function **eCLD_AEAAGetAlertsSend()**. This function returns immediately and the requested alerts are later returned in an E_CLD_APPLIANCE_EVENTS_AND_ALERTS_CMD_GET_ALERTS event, which is generated when a response arrives from the server (Appliance Events and Alerts events are described in Section 47.4).

Note: *The cluster server handles the 'Get Alerts' message automatically and returns the requested alerts in a 'Get Alerts Response' message to the client.*

The appliance alerts from the message payload are contained in the above event - for details of this payload and the alert information, refer to Section 47.8.2. Up to 15 alerts can be reported in a single response.

47.3.2 'Alerts Notification' Messages from Server to Client

The cluster server on the appliance can send unsolicited alert notifications to the client in 'Alerts Notification' messages. The application on the server can send a message of this kind by calling either of the following functions:

- **eCLD_AEAAAlertsNotificationSend()**
- **eCLD_AEAAGetAlertsResponseORAlertsNotificationSend()**

Note: *The latter function is also used internally by the cluster server to send a 'Get Alerts Response' message - see Section 47.3.1.*

The appliance status information from the 'Alerts Notification' message is reported to the application on the cluster client through the event E_CLD_APPLIANCE_EVENTS_AND_ALERTS_CMD_ALERTS_NOTIFICATION, which is generated when the notification arrives from the server (Appliance Events and Alerts events are described in Section 47.4). The appliance alerts from the message payload are contained in the above event - for details of this payload and the alert information, refer to Section 47.8.2. Up to 15 alerts can be reported in a single notification.

47.3.3 'Event Notification' Messages from Server to Client

The cluster server on the appliance can send unsolicited event notifications to the client in 'Event Notification' messages, where each message reports a single appliance event (for example, oven has reached its target temperature). A message of this kind can be sent by the application on the server by calling the function **eCLD_AEAAEventNotificationSend()**.

The appliance event information from the 'Event Notification' message is reported to the application on the cluster client through the event E_CLD_APPLIANCE_EVENTS_AND_ALERTS_CMD_ALERTS_NOTIFICATION, which is generated when the notification arrives from the server (Appliance Events and Alerts events are described in Section 47.4). The appliance event from the message payload is contained in the above client event - for details of this payload and the embedded appliance event information, refer to Section 47.8.3.

47.4 Appliance Events and Alerts Events

The Appliance Events and Alerts cluster has its own events that are handled through the callback mechanism described in Chapter 3. The cluster contains its own event handler. If a device uses this cluster then application-specific Appliance Events and Alerts event handling must be included in the user-defined callback function for the associated endpoint, where this callback function is registered through the relevant endpoint registration function. This callback function is then invoked when an Appliance Events and Alerts event occurs and needs the attention of the application.

For an Appliance Events and Alerts 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 tsCLD_ApplianceEventsAndAlertsCallBackMessage structure:

```
typedef struct
{
  uint8  u8CommandId
  union
  {
  tsCLD_AEAA_GetAlertsResponseORAlertsNotificationPayload
  *psGetAlertsResponseORAlertsNotificationPayload;
  tsCLD_AEAA_EventNotificationPayload
  *psEventNotificationPayload;
  }uMessage;
} tsCLD_ApplianceEventsAndAlertsCallBackMessage;
```

When an Appliance Events and Alerts event occurs, one of four 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 are detailed the tables below for events generated on a server and a client.

Table 110. Appliance Events and Alerts Command Types (Events on Server)

u8CommandId Enumeration	Description
E_CLD_APPLIANCE_EVENTS_AND_ALERTS_CMD_GET_ALERTS	A ‘Get Alerts’ request has been received by the server (appliance)

Table 111. Appliance Events and Alerts Command Types (Events on Client)

u8CommandId Enumeration	Description
E_CLD_APPLIANCE_EVENTS_AND_ALERTS_CMD_GET_ALERTS	A response to a ‘Get Alerts’ request has been received by the client, containing the requested alerts (up to 15)
E_CLD_APPLIANCE_EVENTS_AND_ALERTS_CMD_ALERTS_NOTIFICATION	An ‘Alerts Notification’ message has been received by the client, containing unsolicited alerts (up to 15)
E_CLD_APPLIANCE_EVENTS_AND_ALERTS_CMD_EVENT_NOTIFICATION	An ‘Event Notification’ message has been received by the client

47.5 Functions

The following Appliance Events and Alerts cluster functions are provided:

1. eCLD_ApplianceEventsAndAlertsCreateApplianceEventsAndAlerts
2. eCLD_AEAAGetAlertsSend
3. eCLD_AEAAGetAlertsResponseORAlertsNotificationSend
4. eCLD_AEAAAlertsNotificationSend
5. eCLD_AEAAEventNotificationSend

47.5.1 eCLD_ApplianceEventsAndAlertsCreateApplianceEventsAndAlerts

```
teZCL_Status eCLD_ApplianceEventsAndAlertsCreateApplianceEventsAndAlerts(
 tsZCL_ClusterInstance *psClusterInstance,
 bool_t  bIsServer,
 tsZCL_ClusterDefinition *psClusterDefinition,
 void *pvEndPointSharedStructPtr,
 uint8 *pu8AttributeControlBits,
 tsCLD_ApplianceEventsAndAlertsCustomDataStructure
```

```
            *psCustomDataStructure);
```

Description

This function creates an instance of the Appliance Events and Alerts 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 creates an Appliance Events and Alerts 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 D.

Note: *This function must not be called for an endpoint on which a standard ZigBee device is 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 the ZigBee Devices User Guide (JNUG3131).*

When used, this function must be the first Appliance Events and Alerts cluster function called in the application, and must be called after the stack has been started and after the ZCL has been initialized.

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 is automatically adjusted by the compiler using the following declaration:

```
uint8 au8ApplianceEventsAndAlertsAttributeControlBits
    [(sizeof(asCLD_ApplianceEventsAndAlertsClusterAttributeDefinitions)
    / sizeof(tsZCL_AttributeDefinition))];
```

Parameters

Returns

47.5.2 eCLD_AEAAGetAlertsSend

```
teZCL_Status eCLD_AEAAGetAlertsSend(
        uint8 u8SourceEndPointId,
        uint8 u8DestinationEndPointId,
        tsZCL_Address *psDestinationAddress,
        uint8 *pu8TransactionSequenceNumber);
```

Description

This function can be used on an Appliance Events and Alerts cluster client to send a ‘Get Alerts’ message to a cluster server (appliance).

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the message. The TSN in the response is 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.

Parameter

Returns

47.5.3 eCLD_AEAAGetAlertsResponseORAlertsNotificationSend

```
teZCL_Status eCLD_AEAAGetAlertsResponseORAlertsNotificationSend(
    uint8 u8SourceEndPointId,
    uint8 u8DestinationEndPointId,
    tsZCL_Address *psDestinationAddress,
    uint8 *pu8TransactionSequenceNumber,
    teCLD_ApplianceEventsAndAlerts_CommandId eCommandId,
    tsCLD_AEAA_GetAlertsResponseORAlertsNotificationPayload
 *psPayload);
```

Description

This function can be used on an Appliance Events and Alerts cluster server to send a ‘Get Alerts Response’ message (in reply to a ‘Get Alerts’ message) or an unsolicited ‘Alerts Notification’ message to a cluster client.

The command to be sent must be specified as one of:

- E_CLD_APPLIANCE_EVENTS_AND_ALERTS_CMD_GET_ALERTS
- E_CLD_APPLIANCE_EVENTS_AND_ALERTS_CMD_ALERTS_NOTIFICATION

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the message. The TSN in the response is 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

Returns

47.5.4 eCLD_AEAAAlertsNotificationSend

```
teZCL_Status  eCLD_AEAAAlertsNotificationSend(
 uint8 u8SourceEndPointId,
 uint8 u8DestinationEndPointId,
 tsZCL_Address *psDestinationAddress,
 uint8 *pu8TransactionSequenceNumber,
 tsCLD_AEAA_GetAlertsResponseORAlertsNotificationPayload
 *psPayload);
```

Description

This function can be used on an Appliance Events and Alerts cluster server to send an unsolicited ‘Alerts Notification’ message to a cluster client. The function is an alternative to **eCLD_AEAAGetAlertsResponse ORAlertsNotificationSend()**.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the message. The TSN in the response is 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

Returns

47.5.5 eCLD_AEAAEventNotificationSend

```
teZCL_Status eCLD_AEAAEventNotificationSend(
 uint8 u8SourceEndPointId,
 uint8 u8DestinationEndPointId,
 tsZCL_Address *psDestinationAddress,
 uint8 *pu8TransactionSequenceNumber,
 tsCLD_AEAA_EventNotificationPayload *psPayload);
```

Description

This function can be used on an Appliance Events and Alerts cluster server (appliance) to send an ‘Event Notification’ message to a cluster client, to indicate that an incident has occurred.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the message. The TSN in the response is 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

Returns

47.6 Return codes

The Appliance Events and Alerts cluster functions use the ZCL return codes, listed in Section 7.2.

47.7 Enumerations

47.7.1 ‘Command ID’ Enumerations

The following enumerations are used in commands received on a cluster server or client.

```
typedef enum PACK
{
   E_CLD_APPLIANCE_EVENTS_AND_ALERTS_CMD_GET_ALERTS = 0x00,
   E_CLD_APPLIANCE_EVENTS_AND_ALERTS_CMD_ALERTS_NOTIFICATION,
   E_CLD_APPLIANCE_EVENTS_AND_ALERTS_CMD_EVENT_NOTIFICATION
} teCLD_ApplianceEventsAndAlerts_CommandId;
```

The above enumerations are described in the table below.

Table 112. ‘Command ID’ Enumerations

Enumeration	Description
E_CLD_APPLIANCE_EVENTS_AND_ALERTS_CMD_GET_ALERTS	‘Get Alerts’ request (on server) or response (on client)
E_CLD_APPLIANCE_EVENTS_AND_ALERTS_CMD_ALERTS_NOTIFICATION	Alerts notification (on client)

Table 112. 'Command ID' Enumerations...*continued*

Enumeration	Description
E_CLD_APPLIANCE_EVENTS_AND_ALERTS_CMD_EVENT_NOTIFICATION	Events notification (on server)

47.8 Structures

47.8.1 tsCLD_ApplianceEventsAndAlertsCallBackMessage

For an Appliance Events and Alerts 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 `tsCLD_ApplianceEventsAndAlertsCallBackMessage` structure:

```
typedef struct
{
 uint8     u8CommandId
 union
 {
 tsCLD_AEAA_GetAlertsResponseORAlertsNotificationPayload
         *psGetAlertsResponseORAlertsNotificationPayload;
 tsCLD_AEAA_EventNotificationPayload
         *psEventNotificationPayload;
 } uMessage;
} tsCLD_ApplianceEventsAndAlertsCallBackMessage;
```

where:

- `u8CommandId` indicates the type of Appliance Events and Alerts command that has been received, one of:
  - E_CLD_APPLIANCE_EVENTS_AND_ALERTS_CMD_GET_ALERTS
  - E_CLD_APPLIANCE_EVENTS_AND_ALERTS_CMD_ALERTS_NOTIFICATION
  - E_CLD_APPLIANCE_EVENTS_AND_ALERTS_CMD_EVENT_NOTIFICATION
- `uMessage` is a union containing the command payload as one of (depending on the value of `u8CommandId`):
  - `psGetAlertsResponseORAlertsNotificationPayload` is a pointer to the payload of an "Get Alerts" response message or an alerts notification message (see Section 47.8.2)
  - `psEventNotificationPayload` is a pointer to the payload of an events notification message (see Section 47.8.3)

47.8.2 tsCLD_AEAA_GetAlertsResponseORAlertsNotificationPayload

This structure contains the payload for a 'Get Alerts Response' message or an 'Alerts Notification' message.

```
typedef struct
{
   zuint8  u8AlertsCount;
   zuint24 au24AlertStructure[
             CLD_APPLIANCE_EVENTS_AND_ALERTS_MAXIMUM_NUM_OF_ALERTS];
} tsCLD_AEAA_GetAlertsResponseORAlertsNotificationPayload;
```

where:

- `u8AlertsCount` is an 8-bit bitmap containing the following alerts information:

Bits	Description
0-3	Number of reported alerts

Bits	Description
4-7	Type of alert: • 0x0: Unstructured • 0x1-0xF: Reserved

- `au24AlertStructure[]` is an array of 24-bit bitmaps, with one bitmap for each reported alert, containing the following information:

Bits	Description
0-7	Alert ID: • 0x0: Reserved • 0x01-0x3F: Standardized • 0x40-0x7F: Non-standardized • 0x80-0xFF: Proprietary
8-11	Category: • 0x0: Reserved • 0x1: Warning • 0x2: Danger • 0x3: Failure • 0x4–0xF: Reserved
12-13	Presence or recovery: • 0x0: Presence (alert detected) • 0x1: Recovery (alert recovered) • 0x2–0x3: Reserved
14-15	Reserved (set to 0x0)
16-23	Non-standardized or proprietary

47.8.3 tsCLD_AEAA_EventNotificationPayload

This structure contains the payload for an ‘Event Notification’ message.

```
typedef struct
{
   zuint8           u8EventHeader;
   zuint8           u8EventIdentification;
} tsCLD_AEAA_EventNotificationPayload;
```

where:

- `u8EventHeader` is reserved and set to 0
- `u8EventIdentification` is the identifier of the event being notified:
  - 0x01: End of operational cycle reached
  - 0x02: Reserved
  - 0x03: Reserved
  - 0x04: Target temperature reached
  - 0x05: End of cooking process reached
  - 0x06: Switching off
  - 0xF7: Wrong data

    (Values 0x00 to 0x3F are standardised, 0x40 to 0x7F are non-standardised, and 0x80 to 0xFF except 0xF7 are proprietary)

47.8.4 tsCLD_ApplianceEventsAndAlertsCustomDataStructure

The Appliance Events and Alerts cluster requires extra storage space to be allocated to be used by internal functions. The structure definition for this storage is shown below:

```
typedef struct
{
    tsZCL_ReceiveEventAddress sReceiveEventAddress;
    tsZCL_CallBackEvent sCustomCallBackEvent;
    tsCLD_ApplianceEventsAndAlertsCallBackMessage sCallBackMessage;
} tsCLD_ApplianceEventsAndAlertsCustomDataStructure;
```

The fields are for internal use and no knowledge of them is required.

47.9 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 Appliance Events and Alerts cluster.

To enable the Appliance Events and Alerts cluster in the code to be built, it is necessary to add the following line to the file:

```
#define CLD_APPLIANCE_EVENTS_AND_ALERTS
```

In addition, to enable the cluster as a client or server, it is also necessary to add one of the following lines to the same file:

```
#define APPLIANCE_EVENTS_AND_ALERTS_SERVER
#define APPLIANCE_EVENTS_AND_ALERTS_CLIENT
```

The Appliance Identification cluster contains macros that may be optionally specified at compile-time by adding one or more of the following lines to the **zcl_options.h** file.

Maximum Number of Alerts Reported

The maximum number of alerts that can be reported in a response or notification can be defined (as n) using the following definition in the **zcl_options.h** file:

```
#define CLD_APPLIANCE_EVENTS_AND_ALERTS_MAXIMUM_NUM_OF_ALERTS n
```

The default value is 16, which is the upper limit on this value, and n must therefore not be greater than 16.

Global Attributes

Add this line to define the value (n) of the Cluster Revision attribute:

```
#define CLD_APPLIANCE_EVENTS_AND_ALERTS_CLUSTER_REVISION <n>
```

The default value is 1, which corresponds to the revision of the cluster in the ZCL r6 specification (see Section 2.4).

48 Appliance Statistics Cluster

This chapter outlines the Appliance Statistics cluster, which provides an interface for supplying statistical information about an appliance.

The Appliance Statistics cluster has a Cluster ID of 0x0B03.

48.1 Overview

The Appliance Statistics cluster provides an interface for sending appliance statistics in the form of data logs to a collector node, which may be a gateway.

The cluster is enabled by defining CLD_APPLIANCE_STATISTICS in the **zcl_options.h** file. Further compile-time options for the Appliance Statistics cluster are detailed in Section 48.10.

The cluster client may obtain logs from the server (appliance) in any of the following ways:

- Unsolicited log notifications sent by the server
- Solicited responses obtained by:
  - Client sending ‘Log Queue Request’ to enquire whether logs are available
  - Client sending ‘Log Request’ for each log available
- Semi-solicited responses obtained by:
  - Server sending ‘Statistics Available’ notification to indicate that logs are available
  - Client sending ‘Log Request’ for each log available

48.2 Cluster structure and attributes

The structure definition for the Appliance Statistics cluster (server) is:

```
typedef struct
{
#ifdef APPLIANCE_STATISTICS_SERVER
    zuint32                         u32LogMaxSize;
    zuint8                          u8LogQueueMaxSize;
#endif
    zuint16                         u16ClusterRevision;
} tsCLD_ApplianceStatistics;
```

where:

- `u32LogMaxSize` is a mandatory attribute which specifies the maximum size, in bytes, of the payload of a log notification and log response. This value should not be greater than 70 bytes (otherwise the Partition cluster is needed)
- `u8LogQueueMaxSize` is a mandatory attribute which specifies the maximum number of logs in the queue on the cluster server that are available to be requested by the client
- `u16ClusterRevision` is a mandatory attribute that specifies the revision of the cluster specification on which this cluster instance is based. The cluster specification in the ZCL r6 corresponds to a cluster revision of 1. The value is incremented by one for each subsequent revision of the cluster specification. This attribute is also described in Section 2.4.

48.3 Sending messages

The Appliance Statistics cluster server resides on the appliance (e.g. a washing machine) and the cluster client resides on a controlling device (normally a remote control unit).

Messages can be sent between the client and the server in the following ways:

- The client can enquire whether any data logs are available on the appliance (server) by sending a ‘Log Queue Request’ to the server (which will reply with a ‘Log Queue Response’ message)
- The server can notify the client that data logs are available by sending an unsolicited ‘Statistics Available’ message to the client
- The client can request a current data log from the appliance (server) by sending a ‘Log Request’ message to the server (which will reply with a ‘Log Response’ message)
- The server can send an unsolicited data log to the client in a ‘Log Notification’ message

Sending the above messages is described in the sub-sections below.

48.3.1 ‘Log Queue Request’ messages from client to server

The application on the cluster client can enquire about the availability of data logs on the appliance by sending a ‘Log Queue Request’ message to the server. This message is sent by calling the function **eCLD_ASCLogQueueRequestSend()**. This function returns immediately and the log availability is later returned in an E_CLD_APPLIANCE_STATISTICS_CMD_LOG_QUEUE_RESPONSE event, which is generated when a response arrives from the server (Appliance Statistics events are described in Section 48.5).

Note: *The cluster server handles the ‘Log Queue Request’ message automatically and returns the requested information in a ‘Log Queue Response’ message to the client.*

The log availability information from the message payload is contained in the above event, and comprises the number of logs currently in the log queue and their log IDs - for details of this payload and the availability information, refer to Section 48.9.4.

48.3.2 ‘Statistics Available’ messages from server to client

The cluster server can notify the client when data logs are available by sending an unsolicited ‘Statistics Available’ message to the client. This message contains the number of logs in the log queue and the log IDs. A message of this kind can be sent by the application on the server by calling either of the following functions:

- **eCLD_ASCStatisticsAvailableSend()**
- **eCLD_ASCLogQueueResponseORStatisticsAvailableSend()**

Note:

1. *The latter function is also used internally by the cluster server to send a ‘Log Queue Response’ message - see Section 48.3.1.*
2. *Before calling either function, the relevant log(s) should be added to the local log queue as described in Section 48.4.1. This is because the logs need to be in the queue to allow the server to perform further actions on them - for example, to process a ‘Log Request’.*

The log availability information from the ‘Statistics Available’ message is reported to the application on the cluster client through the event E_CLD_APPLIANCE_STATISTICS_CMD_STATISTICS_AVAILABLE, which is generated when the message arrives from the server (Appliance Statistics events are described in Section 48.5). The availability information from the message payload is contained in the above event - for details of this payload and the availability information, refer to Section 48.9.4.

48.3.3 ‘Log Request’ messages from client to server

The application on the cluster client can request the log with a particular log ID from the appliance by sending a ‘Log Request’ message to the server. This message is sent by calling the function **eCLD_ASCLogRequestSend()**. This function returns immediately and the requested log information is later returned in an E_CLD_APPLIANCE_STATISTICS_CMD_LOG_QUEUE_RESPONSE event, which is generated when a response arrives from the server (Appliance Statistics events are described in Section 48.5).

Note:

1. *This function should normally be called after a 'Log Queue Response' or 'Statistics Available' message has been received by the client, indicating that logs are available on the server.*
2. *The cluster server handles the 'Log Request' message automatically and returns the requested log information in a 'Log Response' message to the client.*

The log information from the message payload is contained in the above event - for details of this payload and the supplied log information, refer to Section 48.9.3.

48.3.4 'Log Notification' messages from server to client

The cluster server can supply the client with an individual data log by sending an unsolicited 'Log Notification' message to the client. This message is sent by the application on the server by calling either of the following functions:

- **eCLD_ASCLogNotificationSend()**
- **eCLD_ASCLogNotificationORLogResponseSend()**

Note:

1. *The latter function is also used internally by the cluster server to send a 'Log Response' message - see Section 48.3.1.*
2. *Before calling either function, the relevant log should be in the local log queue (see Section 48.4.1). This is because the log needs to be in the queue to allow the server to perform further actions on it - for example, to process a 'Log Request'.*
3. *The function **eCLD_ASCAddLog()** used to add a log to the local log queue (see Section 48.4.1) automatically sends a 'Log Notification' message to all bound Appliance Statistics cluster clients.*

The log information from the 'Log Notification' message is reported to the application on the cluster client through the event E_CLD_APPLIANCE_STATISTICS_CMD_LOG_NOTIFICATION, which is generated when the message arrives from the server (Appliance Statistics events are described in Section 48.5). The log information from the message payload is contained in the above event - for details of this payload and the supplied log information, refer to Section 48.9.3.

48.4 Log Operations on Server

Appliance Statistics cluster functions are provided to allow the application on the cluster server (appliance) to perform the following local log operations:

- Add a log to the log queue
- Remove a log from the log queue
- Obtain a list of the logs in the log queue
- Obtain an individual log from the log queue

These operations are described in the sub-sections below.

48.4.1 Adding and Removing Logs

A data log can be added to the local log queue (on the cluster server) using the function **eCLD_ASCAddLog()**. The log must be given an identifier and the UTC time at which the log was added must be specified. The length of the log, in bytes, must be less than the value of CLD_APPLIANCE_STATISTICS_ATTR_LOG_MAX_SIZE, which is defined in the **zcl_options.h** files (and must be less than or equal to 70).

The above function also sends a 'Log Notification' message to all bound Appliance Statistics cluster clients.

An existing log can be removed from the local log queue using the function **eCLD_ASCRemoveLog()**. The log is specified using its identifier.

48.4.2 Obtaining Logs

A list of the logs that are currently in the local log queue (on the cluster server) can be obtained by calling the function **eCLD_ASCGetLogsAvailable()**. This function provides the number of logs in the queue and a list of the log identifiers.

An individual log from the local log queue can be obtained using the function **eCLD_ASCGetLogEntry()**. The required log is specified by means of its identifier.

Normally, **eCLD_ASCGetLogsAvailable()** is called first to obtain a list of the available logs and then **eCLD_ASCGetLogEntry()** is called for each log.

48.5 Appliance statistics events

The Appliance Statistics cluster has its own events that are handled through the callback mechanism described in Chapter 3. The cluster contains its own event handler. If a device uses this cluster then application-specific Appliance Statistics event handling must be included in the user-defined callback function for the associated endpoint, where this callback function is registered through the relevant endpoint registration function. This callback function will then be invoked when an Appliance Statistics event occurs and needs the attention of the application.

For an Appliance Statistics 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 `tsCLD_ApplianceStatisticsCallBackMessage` structure:

```
typedef struct
{
    uint8    u8CommandId;
    union
    {
        tsCLD_ASC_LogNotificationORLogResponsePayload
                                        *psLogNotificationORLogResponsePayload;
        tsCLD_ASC_LogQueueResponseORStatisticsAvailablePayload

 *psLogQueueResponseORStatisticsAvailabePayload;
        tsCLD_ASC_LogRequestPayload   *psLogRequestPayload;
    } uMessage;
} tsCLD_ApplianceStatisticsCallBackMessage;
```

When an Appliance Statistics event occurs, one of four 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 are detailed the tables below for events generated on a server and a client.

Table 113. Appliance Statistics Command Types (Events on Server)

u8CommandId Enumeration	Description
E_CLD_APPLIANCE_STATISTICS_ CMD_LOG_REQUEST	A ‘Log Request’ message has been received by the server (appliance)
E_CLD_APPLIANCE_STATISTICS_ CMD_LOG_QUEUE_REQUEST	A ‘Log Queue Request’ message has been received by the server (appliance)

Table 114. Appliance Statistics Command Types (Events on Client)

u8CommandId Enumeration	Description
E_CLD_APPLIANCE_STATISTICS_CMD_LOG_NOTIFICATION	A 'Log Notification' message has been received by the client
E_CLD_APPLIANCE_STATISTICS_CMD_LOG_RESPONSE	A 'Log Response' message has been received by the client
E_CLD_APPLIANCE_STATISTICS_CMD_LOG_QUEUE_RESPONSE	A 'Log Queue Response' message has been received by the client
E_CLD_APPLIANCE_STATISTICS_CMD_STATISTICS_AVAILABLE	A 'Statistics Available' message has been received by the client

48.6 Functions

The following Appliance Statistics cluster functions are provided:

1. eCLD_ApplianceStatisticsCreateApplianceStatistics
2. eCLD_ASCAddLog
3. eCLD_ASCRemoveLog
4. eCLD_ASCGetLogsAvailable
5. eCLD_ASCGetLogEntry
6. eCLD_ASCLogQueueRequestSend
7. eCLD_ASCLogRequestSend
8. eCLD_ASCLogQueueResponseORStatisticsAvailableSend
9. eCLD_ASCStatisticsAvailableSend
10. eCLD_ASCLogNotificationORLogResponseSend
11. eCLD_ASCLogNotificationSend

48.6.1 eCLD_ApplianceStatisticsCreateApplianceStatistics

```
teZCL_Status eCLD_ApplianceStatisticsCreateApplianceStatistics(
        tsZCL_ClusterInstance *psClusterInstance,
        bool_t bIsServer,
        tsZCL_ClusterDefinition *psClusterDefinition,
        void *pvEndPointSharedStructPtr,
        uint8 *pu8AttributeControlBits,
        tsCLD_ApplianceStatisticsCustomDataStructure
 *psCustomDataStructure);
```

Description

This function creates an instance of the Appliance Statistics 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 an Appliance Statistics 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 D.

Note: *This function must not be called for an endpoint on which a standard ZigBee device is 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 the ZigBee Devices User Guide*

Note: *(JNUG3131).*

When used, this function must be the first Appliance Statistics cluster function called in the application, and must be called after the stack has been started and after the ZCL has been initialized.

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 is automatically adjusted by the compiler using the following declaration:

```
uint8 au8ApplianceStatisticsAttributeControlBits
[(sizeof(asCLD_ApplianceStatisticsClusterAttributeDefinitions) /
 sizeof(tsZCL_AttributeDefinition))];
```

Parameters

- *psClusterInstance* Pointer to structure containing information about the cluster instance to be created (see Section 6.1.16). This structure is updated by the function by initialising individual structure fields.
- *bIsServer* Type of cluster instance (server or client) to be created:
- TRUE - server
- FALSE - client
- *psClusterDefinition* Pointer to structure indicating the type of cluster to be created (see see Section 6.1.2). In this case, this structure must contain the details of the Appliance Statistics cluster. This parameter can refer to a pre-filled structure called `sCLD_ApplianceStatistics` which is provided in the **ApplianceStatistics.h** file.
- *pvEndPointSharedStructPtr* Pointer to the shared structure used for attribute storage. This parameter should be the address of the structure of type `tsCLD_ApplianceStatistics` which defines the attributes of Appliance Statistics cluster. The function initializes the attributes with default values.
- *pu8AttributeControlBits* Pointer to an array of **uint8** values, with one element for each attribute in the cluster (see above). For a cluster client, set this pointer to NULL.
- *psCustomDataStructure* Pointer to a structure containing the storage for internal functions of the cluster (see Section 48.9.6).

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_INVALID_VALUE

48.6.2 eCLD_ASCAddLog

```
teZCL_CommandStatus eCLD_ASCAddLog(
        uint8 u8SourceEndPointId,
        uint32 u32LogId,
        uint8 u8LogLength,
        uint32 u32Time,
                uint8 *pu8LogData);
```

Description

This function can be used on an Appliance Statistics cluster server to add a data log to the log queue. The function also sends out a 'Log Notification' message to all bound Appliance Statistics cluster clients.

The length of the data log, in bytes, must be less than the defined value of CLD_APPLIANCE_STATISTICS_ATTR_LOG_MAX_SIZE (which must be less than or equal to 70).

Parameter

- *u8SourceEndPointId* Number of the local endpoint on which the Appliance Statistics cluster server resides
- *u32LogId* Identifier of log
- *u8LogLength* Length of log, in bytes
- *u32Time* UTC time at which log was produced
- *pu8LogData* Pointer to log data

Returns

- E_ZCL_CMDS_SUCCESS
- E_ZCL_CMDS_FAIL
- E_ZCL_CMDS_INVALID_VALUE (log too long)
- E_ZCL_CMDS_INVALID_FIELD (NULL pointer to log data)
- E_ZCL_CMDS_INSUFFICIENT_SPACE

48.6.3 eCLD_ASCRemoveLog

```
teZCL_CommandStatus eCLD_ASCRemoveLog(
                uint8 u8SourceEndPointId,
                uint32 u32LogId);
```

Description

This function can be used on an Appliance Statistics cluster server to remove the specified data log from the log queue.

Parameter

- *u8SourceEndPointId* Number of the local endpoint on which the Appliance Statistics cluster server resides
- *u32LogId* Identifier of log

Returns

- E_ZCL_CMDS_SUCCESS
- E_ZCL_CMDS_FAIL

48.6.4 eCLD_ASCGetLogsAvailable

```
teZCL_CommandStatus eCLD_ASCGetLogsAvailable(
        uint8 u8SourceEndPointId,
        uint32 *pu32LogId,
        uint8 *pu8LogIdCount);
```

Description

This function can be used on an Appliance Statistics cluster server to obtain a list of the data logs in the log queue. The number of available logs and a list of their log IDs will be obtained.

Parameter

- *u8SourceEndPointId* Number of the local endpoint on which the Appliance Statistics cluster server resides
- *pu32LogId* Pointer to an area of memory to receive the list of 32-bit log IDs
- *pu8LogIdCount* Pointer to an area of memory to receive the number of logs in the queue

Returns

- E_ZCL_CMDS_SUCCESS
- E_ZCL_CMDS_FAIL

48.6.5 eCLD_ASCGetLogEntry

```
teZCL_CommandStatus eCLD_ASCGetLogEntry(
        uint8 u8SourceEndPointId,
        uint32 u32LogId,
        tsCLD_LogTable **ppsLogTable);
```

Description

This function can be used on an Appliance Statistics cluster server to obtain the data log with the specified log ID.

Parameter

- *u8SourceEndPointId* Number of the local endpoint on which the Appliance Statistics cluster server resides
- *u32LogId* Log ID of the required data log
- *ppsLogTable* Pointer to a memory location to receive a pointer to the required data log

Returns

- E_ZCL_CMDS_SUCCESS
- E_ZCL_CMDS_FAIL
- E_ZCL_CMDS_NOT_FOUND (specified log not present)

48.6.6 eCLD_ASCLogQueueRequestSend

```
teZCL_Status eCLD_ASCLogQueueRequestSend(
        uint8 u8SourceEndPointId,
        uint8 u8DestinationEndPointId,
        tsZCL_Address *psDestinationAddress,
        uint8 *pu8TransactionSequenceNumber);
```

Description

This function can be used on an Appliance Statistics cluster client to send a ‘Log Queue Request’ message to a cluster server (appliance), in order enquire about the availability of logs on the server.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the message. The TSN in the response is 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 to send the message. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values
- *u8DestinationEndPointId* Number of the endpoint on the remote node to which the message is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP
- *psDestinationAddress* Pointer to a structure holding the address of the node to which the message is sent
- *pu8TransactionSequenceNumber* Pointer to a location to receive the Transaction Sequence Number (TSN) of the message

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_INVALID_VALUE

48.6.7 eCLD_ASCLogRequestSend

```
teZCL_Status eCLD_ASCLogRequestSend(
        uint8 u8SourceEndPointId,
        uint8 u8DestinationEndPointId,
        tsZCL_Address *psDestinationAddress,
        uint8 *pu8TransactionSequenceNumber,
        tsCLD_ASC_LogRequestPayload *psPayload);
```

Description

This function can be used on an Appliance Statistics cluster client to send a ‘Log Request’ message to a cluster server (appliance), in order request the data log with a specified log ID.

The function should normally be called after enquiring about log availability using the function **eCLD_ASCLogQueueRequestSend()** or after receiving an unsolicited ‘Statistics Available’ notification from the server.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the message. The TSN in the response is 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 to send the message. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values

- *u8DestinationEndPointId* Number of the endpoint on the remote node to which the message is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP
- *psDestinationAddress* Pointer to a structure holding the address of the node to which the message is sent
- *pu8TransactionSequenceNumber* Pointer to a location to receive the Transaction Sequence Number (TSN) of the message
- *psPayload* Pointer to *a* structure containing the payload for the 'Log Request', including the relevant log ID (see Section 48.9.2)

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_INVALID_VALUE

48.6.8 eCLD_ASCLogQueueResponseORStatisticsAvailableSend

```
teZCL_Status eCLD_ASCLogQueueResponseORStatisticsAvailableSend(
        uint8 u8SourceEndPointId,
        uint8 u8DestinationEndPointId,
        tsZCL_Address *psDestinationAddress,
        uint8 *pu8TransactionSequenceNumber,
        teCLD_ApplianceStatistics_ServerCommandId
 eCommandId);
```

Description

This function can be used on an Appliance Statistics cluster server to send a 'Log Queue Response' message (in reply to a 'Log Queue Request' message) or an unsolicited 'Statistics Available' message to a cluster client.

The command to be sent must be specified as one of:

- E_CLD_APPLIANCE_STATISTICS_CMD_LOG_QUEUE_RESPONSE
- E_CLD_APPLIANCE_STATISTICS_CMD_STATISTICS_AVAILABLE

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the message. The TSN in the response is 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 to send the message. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values
- *u8DestinationEndPointId* Number of the endpoint on the remote node to which the message is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP
- *psDestinationAddress* Pointer to a structure holding the address of the node to which the message is sent
- *pu8TransactionSequenceNumber* Pointer to a location to receive the Transaction Sequence Number (TSN) of the message
- *eCommandId* Enumeration indicating the command to be sent (see above and Section 48.8.3)

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_INVALID_VALUE

48.6.9 eCLD_ASCStatisticsAvailableSend

```
teZCL_Status eCLD_ASCStatisticsAvailableSend(
        uint8 u8SourceEndPointId,
        uint8 u8DestinationEndPointId,
        tsZCL_Address *psDestinationAddress,
        uint8 *pu8TransactionSequenceNumber);
```

Description

This function can be used on an Appliance Statistics cluster server to send an unsolicited ‘Statistics Available’ message to a cluster client. The function is an alternative to **eCLD_ASCLogQueueResponseORStatisticsAvailableSend()**.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the message. The TSN in the response is 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 to send the message. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values
- *u8DestinationEndPointId* Number of the endpoint on the remote node to which the message is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP
- *psDestinationAddress* Pointer to a structure holding the address of the node to which the message is sent
- *pu8TransactionSequenceNumber* Pointer to a location to receive the Transaction Sequence Number (TSN) of the message

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_INVALID_VALUE

48.6.10 eCLD_ASCLogNotificationORLogResponseSend

```
teZCL_Status eCLD_ASCLogNotificationORLogResponseSend(
        uint8 u8SourceEndPointId,
        uint8 u8DestinationEndPointId,
        tsZCL_Address *psDestinationAddress,
        uint8 *pu8TransactionSequenceNumber,
        teCLD_ApplianceStatistics_ServerCommandId
 eCommandId,
```

```
        tsCLD_ASC_LogNotificationORLogResponsePayload
 *psPayload);
```

Description

This function can be used on an Appliance Statistics cluster server to send a ‘Log Response’ message (in reply to a ‘Log Request’ message) or an unsolicited ‘Log Notification’ message to a cluster client.

The command to be sent must be specified as one of:

- E_CLD_APPLIANCE_STATISTICS_CMD_LOG_NOTIFICATION
- E_CLD_APPLIANCE_STATISTICS_CMD_LOG_RESPONSE

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the message. The TSN in the response is 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 to send the message. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values
- *u8DestinationEndPointId* Number of the endpoint on the remote node to which the message is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP
- *psDestinationAddress* Pointer to a structure holding the address of the node to which the message is sent
- *pu8TransactionSequenceNumber* Pointer to a location to receive the Transaction Sequence Number (TSN) of the message
- *eCommandId* Enumeration indicating the command to be sent (see above and Section 48.8.3)
- *psPayload* Pointer to structure containing payload for message (see Section 48.9.3)

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_INVALID_VALUE

48.6.11 eCLD_ASCLogNotificationSend

```
teZCL_Status eCLD_ASCLogNotificationSend(
        uint8 u8SourceEndPointId,
        uint8 u8DestinationEndPointId,
        tsZCL_Address *psDestinationAddress,
        uint8 *pu8TransactionSequenceNumber,
        tsCLD_ASC_LogNotificationORLogResponsePayload
 *psPayload);
```

Description

This function can be used on an Appliance Statistics cluster server to send an unsolicited ‘Log Notification’ message to a cluster client. The function is an alternative to **eCLD_ASCLogNotificationORLogResponseSend()**.

You are required to provide a pointer to a location to receive a Transaction Sequence Number (TSN) for the message. The TSN in the response is 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 to send the message. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values
- *u8DestinationEndPointId* Number of the endpoint on the remote node to which the message is sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP
- *psDestinationAddress* Pointer to a structure holding the address of the node to which the message is sent
- *pu8TransactionSequenceNumber* Pointer to a location to receive the Transaction Sequence Number (TSN) of the message
- *psPayload* Pointer to structure containing payload for message (see Section 48.9.3)

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL
- E_ZCL_ERR_PARAMETER_NULL
- E_ZCL_ERR_INVALID_VALUE

48.7 Return codes

The Appliance Statistics cluster functions use the ZCL return codes, listed in Section 7.2.

48.8 Enumerations

48.8.1 'Attribute ID' enumerations

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

```
typedef enum PACK
{
    E_CLD_APPLIANCE_STATISTICS_ATTR_ID_LOG_MAX_SIZE = 0x0000,
    E_CLD_APPLIANCE_STATISTICS_ATTR_ID_LOG_QUEUE_MAX_SIZE
} teCLD_ApplianceStatistics_Cluster_AttrID;
```

48.8.2 'Client Command ID' enumerations

The following enumerations are used in commands issued on a cluster client.

```
typedef enum PACK
{
    E_CLD_APPLIANCE_STATISTICS_CMD_LOG_REQUEST = 0x00,
    E_CLD_APPLIANCE_STATISTICS_CMD_LOG_QUEUE_REQUEST
} teCLD_ApplianceStatistics_ClientCommandId;
```

The above enumerations are described in the table below.

Table 115. ‘Client Command ID’ Enumerations

Enumeration	Description
E_CLD_APPLIANCE_STATISTICS_CMD_LOG_REQUEST	‘Log Request’ message
E_CLD_APPLIANCE_STATISTICS_CMD_LOG_QUEUE_REQUEST	‘Log Queue Request’ message

48.8.3 ‘Server Command ID’ enumerations

The following enumerations are used in commands issued on a cluster server.

```
typedef enum PACK
{
    E_CLD_APPLIANCE_STATISTICS_CMD_LOG_NOTIFICATION = 0x00,
    E_CLD_APPLIANCE_STATISTICS_CMD_LOG_RESPONSE,
    E_CLD_APPLIANCE_STATISTICS_CMD_LOG_QUEUE_RESPONSE,
    E_CLD_APPLIANCE_STATISTICS_CMD_STATISTICS_AVAILABLE
} teCLD_ApplianceStatistics_ServerCommandId;
```

The above enumerations are described in the table below.

Table 116. ‘Server Command ID’ Enumerations

Enumeration	Description
E_CLD_APPLIANCE_STATISTICS_CMD_LOG_NOTIFICATION	A ‘Log Notification’ message
E_CLD_APPLIANCE_STATISTICS_CMD_LOG_RESPONSE	A ‘Log Response’ message
E_CLD_APPLIANCE_STATISTICS_CMD_LOG_QUEUE_RESPONSE	A ‘Log Queue Response’ message
E_CLD_APPLIANCE_STATISTICS_CMD_STATISTICS_AVAILABLE	A ‘Statistics Available’ message

48.9 Structures

48.9.1 tsCLD_ApplianceStatisticsCallBackMessage

For an Appliance Statistics 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 `tsCLD_ApplianceStatisticsCallBackMessage` structure:

```
typedef struct
{
    uint8    u8CommandId;
    union
    {
       tsCLD_ASC_LogNotificationORLogResponsePayload
                                  *psLogNotificationORLogResponsePayload;
       tsCLD_ASC_LogQueueResponseORStatisticsAvailablePayload

 *psLogQueueResponseORStatisticsAvailabePayload;
       tsCLD_ASC_LogRequestPayload *psLogRequestPayload;
    } uMessage;
} tsCLD_ApplianceStatisticsCallBackMessage;
```

where:

- `u8CommandId` indicates the type of Appliance Statistics command that has been received, one of:
  - E_CLD_APPLIANCE_STATISTICS_CMD_LOG_REQUEST

- E_CLD_APPLIANCE_STATISTICS_CMD_LOG_QUEUE_REQUEST
- E_CLD_APPLIANCE_STATISTICS_CMD_LOG_NOTIFICATION
- E_CLD_APPLIANCE_STATISTICS_CMD_LOG_RESPONSE
- E_CLD_APPLIANCE_STATISTICS_CMD_LOG_QUEUE_RESPONSE
- E_CLD_APPLIANCE_STATISTICS_CMD_STATISTICS_AVAILABLE

- `uMessage` is a union containing the command payload as one of (depending on the value of `u8CommandId`):
  - `psLogNotificationORLogResponsePayload` is a pointer to the payload of a 'Log Notification' or 'Log Response' message (see Section 48.9.3)
  - `psLogQueueResponseORStatisticsAvailabePayload` is a pointer to the payload of a 'Log Queue Response' or 'Statistics Available' message (see Section 48.9.4)
  - `psLogRequestPayload` is a pointer to the payload of a 'Log Request' message (see Section 48.9.2)

48.9.2 tsCLD_ASC_LogRequestPayload

This structure contains the payload for the 'Log Request' message.

```
typedef struct
{
    zuint32   u32LogId;
} tsCLD_ASC_LogRequestPayload;
```

where `u32LogId` is the identifier of the data log being requested.

48.9.3 tsCLD_ASC_LogNotificationORLogResponsePayload

This structure contains the payload for the 'Log Notification' and 'Log Response' messages.

```
typedef struct
{
    zutctime    utctTime;
    zuint32     u32LogId;
    zuint32     u32LogLength;
    uint8      *pu8LogData;
} tsCLD_ASC_LogNotificationORLogResponsePayload;
```

where:

- `utctTime` is the UTC time at which the reported log was produced
- `u32LogId` is the identifier of the reported log
- `u32LogLength` is the length, in bytes, of the reported log
- `pu8LogData` is a pointer to an area of memory to receive the data of the reported log

48.9.4 tsCLD_ASC_LogQueueResponseORStatisticsAvailablePayload

This structure contains the payload for the 'Log Queue Response' and 'Statistics Available' messages.

```
typedef struct
{
    zuint8     u8LogQueueSize;
    zuint32   *pu32LogId;
} tsCLD_ASC_LogQueueResponseORStatisticsAvailablePayload;
```

where:

- u8LogQueueSize indicates the number of logs currently in the log queue
- pu32LogId is a pointer to an area of memory to receive the sequence of 32-bit log IDs of the logs in the queue

48.9.5 tsCLD_LogTable

This structure is used to store the details of a data log.

```
typedef struct
{
    zutctime    utctTime;
    uint32      u32LogID;
    uint8       u8LogLength;
    uint8      *pu8LogData;
} tsCLD_LogTable;
```

where:

- utctTime is the UTC time at which the log was produced
- u32LogId is the identifier of the log
- u32LogLength is the length, in bytes, of the log
- pu8LogData is a pointer to an area of memory to receive the data of the log

48.9.6 tsCLD_ApplianceStatisticsCustomDataStructure

The Appliance Statistics cluster requires extra storage space to be allocated to be used by internal functions. The structure definition for this storage is shown below:

```
typedef struct
{
    tsZCL_ReceiveEventAddress sReceiveEventAddress;
    tsZCL_CallBackEvent sCustomCallBackEvent;
    tsCLD_ApplianceStatisticsCallBackMessage sCallBackMessage;
#if (defined CLD_APPLIANCE_STATISTICS) && (defined APPLIANCE_STATISTICS_SERVER)
    tsCLD_LogTable asLogTable[CLD_APPLIANCE_STATISTICS_ATTR_LOG_QUEUE_MAX_SIZE];
#endif
} tsCLD_ApplianceStatisticsCustomDataStructure;
```

The fields are for internal use and no knowledge of them is required.

48.10 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 Appliance Statistics cluster.

To enable the Appliance Statistics cluster in the code to be built, it is necessary to add the following line to the file:

```
#define CLD_APPLIANCE_STATISTICS
```

In addition, to enable the cluster as a client or server, it is also necessary to add one of the following lines to the same file:

```
#define APPLIANCE_STATISTICS_SERVER
#define APPLIANCE_STATISTICS_CLIENT
```

The Appliance Statistics cluster contains macros that may be optionally specified at compile-time by adding some or all the following lines to the **zcl_options.h** file.

Global Attributes

Add this line to define the value (n) of the Cluster Revision attribute:

```
#define CLD_APPLIANCE_STATISTICS_CLUSTER_REVISION <n>
```

The default value is 1, which corresponds to the revision of the cluster in the ZCL r6 specification (see Section 2.4).

Maximum Log Size

Add this line to configure the maximum size n, in bytes, of a data log:

```
#define CLD_APPLIANCE_STATISTICS_ATTR_LOG_MAX_SIZE n
```

The default value is 70 bytes, which is the upper limit on this value, and n must therefore not be greater than 70.

The same value must be defined on the cluster server and client.

Maximum Log Queue Length

Add this line to configure the maximum number of logs n in a log queue:

```
#define CLD_APPLIANCE_STATISTICS_ATTR_LOG_QUEUE_MAX_SIZE n
```

The default value is 15, which is the upper limit on this value, and n must therefore not be greater than 15.

The same value must be defined on the cluster server and client.

Enable Insertion of UTC Time

Add this line to enable the application to insert UTC time data into logs:

```
#define CLD_APPLIANCE_STATISTICS_ATTR_LOG_QUEUE_MAX_SIZE n
```

Disable APS Acknowledgements for Bound Transmissions

Add this line to disable APS acknowledgements for bound transmissions from this cluster:

```
#define CLD_ASC_BOUND_TX_WITH_APS_ACK_DISABLED
```

Part XII: Over-The-Air Upgrade

This part comprises only one chapter:

- Chapter 49 details the **OTA (Over-the-Air) Upgrade** cluster

49 OTA Upgrade cluster

This chapter describes the Over-The-Air (OTA) Upgrade cluster. The OTA Upgrade cluster has a Cluster ID of 0x0019.

Note: *This chapter assumes that the ZigBee 3.0 network consists of nodes which contain only one processor - such as a JN518x, K32W041, K32W061, MCXW71, or MCXW72 microcontroller. However, the OTA Upgrade cluster can also be used with dual-processor nodes (containing a JN518x, K32W041, K32W061, MCXW71, MCXW72 device and a coprocessor), as described in Appendix F.*

49.1 Overview

The Over-The-Air (OTA) Upgrade cluster provides the facility to upgrade (or downgrade or re-install) application software on the nodes of a ZigBee PRO network by:

1. Distributing the replacement software through the network (over the air) from a designated node.

2. Updating the software in a node with minimal interruption to the operation of the node.

The OTA Upgrade cluster acts as a server on the node that distributes the software and as a client on the nodes that receive software updates from the server. The cluster server receives the software from outside the network.

An application that uses the OTA Upgrade cluster must include the header files **zcl_options.h** and **OTA.h**.

The OTA Upgrade cluster is enabled by defining CLD_OTA in the **zcl_options.h** file. Further compile-time options for the OTA Upgrade cluster are detailed in Section 49.13.

When including the OTA Upgrade facility in your application, you should increase the CPU stack size from the default value (as described in Section 49.5).

49.2 OTA Upgrade Images in Internal Flash Memory

This section provides guidance on how to organize Over-The-Air (OTA) upgrade images in internal Flash memory on the target node (OTA Upgrade client). The OTA Upgrade cluster is described in Chapter 49.

By default, OTA upgrade images are downloaded to a Flash memory device that is internal to the device of the OTA Upgrade client. However, the images can optionally be downloaded directly to devices internal Flash memory - this is enabled using the compile-time option, OTA_INTERNAL_STORAGE (see Section 49.13).

The function **eOTA_AllocateEndpointOTASpace()** is used in the application to allocate locations in Flash memory to store application images as part of the OTA upgrade process. The OTA code then uses these locations to store the upgrade image before switching to it, after validation.

There are two issues relating to OTA upgrade and Flash memory remapping:

- Whether the size of the OTA upgrade binary file is larger than the previous version, such that it must use another sector
- Where in the memory space the OTA image is written to

Consider the following cases.

We have a 154 KB image (5 sectors) and download a new image of the same size, starting at sector 8:

Logical Sector	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Physical Sector	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Contents	Running image	Empty	New image	Empty

When we switch to the new image, the physical sectors are moved in the memory map by the bootloader so that the new image becomes the running image and the previous running image becomes the old image. Only the sectors that must be moved are actually moved by the bootloader, and the other sectors are left alone:

Logical Sector	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Physical Sector	8	9	10	11	12	5	6	7	0	1	2	3	4	13	14	15
Contents	New running image					Empty			Old image					Empty		

If we now download a further new image that is 161KB (6 sectors) in size, it would replace the old running image and be placed into logical sectors 8 to 13, which are physical sectors 0, 1, 2, 3, 4 and 13. However, this new image is then unusable because the physical sectors are not contiguous and the bootloader does not take this into account when it remaps the memory (if the new image was less than 160KB, there would be no problem).

The simple solution to this problem is to replace the remapping that the bootloader has chosen with our own remapping in which logical sector 13 becomes physical sector 5, thus allowing the new image to be stored in contiguous physical sectors (0 to 5).

Logical Sector	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Physical Sector	8	9	10	11	12	13	14	15	0	1	2	3	4	5	6	7
Contents	Running image					Empty			New image						Empty	

However, this does not leave any space for permanent data and it also assumes that the new image is stored at logical sector 8.

You may choose to put the new image anywhere in the Flash memory (ZigBee allows this to be configured, and a user-developed solution is free to do what it requires). So you need to adjust the remapping to match. For example, if the OTA image was placed at logical sector 7:

Logical Sector	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Physical Sector	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Contents	Running image					Empty		New image					Empty			

The purpose of this is to leave some sectors at the end of Flash memory for permanent data (otherwise you could always start the OTA image at sector 8).

In such cases, the sensible approach is to:

1. Calculate how much permanent data space is required and reserve the end sectors for this data.
2. Divide the remaining space into two equal blocks of sectors
3. Configure the OTA upgrade to start at the beginning of the second block of sectors.
4. Force the remapping to swap the two blocks, regardless of the actual image size.

Consider the example in which a user wants 64KB for permanent data, which requires 2 sectors. This leaves 14 sectors for applications, so we have two blocks of 7 sectors for each application (even though the application may be smaller than this):

Logical Sector	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Physical Sector	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Contents	Running image					Empty		New image					Empty		Data	

To avoid any problem with the new image growing and needing 6 sectors rather than 5 sectors, we force the remapping to swap all 7 sectors over:

Logical Sector	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Physical Sector	7	8	9	10	11	12	13	0	1	2	3	4	5	6	14	15
Contents	New running image					Empty		Old image					Empty		Data	

This leaves sectors 14 and 15 in a fixed location.

The code to achieve this is as follows:

```
    if (u8CurrentImageSector > 0)
    {
       /* Remapping will not affect the current running image,
          which was already running in a continuous block at the base
          application Flash address */
       vREG_SysWrite(REG_SYS_FLASH_REMAP,  0x0dcba987);
       vREG_SysWrite(REG_SYS_FLASH_REMAP2, 0xfe654321);
    }
```

49.3 OTA Upgrade Cluster structure and attributes

The attributes of the OTA Upgrade cluster are contained in the following structure, which is located only on cluster clients:

```
typedef struct
{
#ifdef OTA_CLIENT
    uint64 u64UgradeServerID;
    uint32 u32FileOffset;
    uint32 u32CurrentFileVersion;
    uint16 u16CurrentStackVersion;
    uint32 u32DownloadedFileVersion;
    uint16 u16DownloadedStackVersion;
    uint8  u8ImageUpgradeStatus;
    uint16 u16ManfId;
    uint16 u16ImageType;
    uint16 u16MinBlockRequestDelay;
#endif
    uint16 u16ClusterRevision;
} tsCLD_AS_Ota;
```

where:

- `u64UgradeServerID` contains the 64-bit IEEE/MAC address of the OTA Upgrade server for the client. This address can be fixed during manufacture or discovered during network formation/operation. If not pre-set, the default value is 0xFFFFFFFFFFFFFFFF. This attribute is mandatory.
- `u32FileOffset` contains the start address in local Flash memory of the upgrade image (that may be currently in transfer from server to client). This attribute is optional.
- `u32CurrentFileVersion` contains the file version of the firmware currently running on the client. This attribute is optional.
- `u16CurrentStackVersion` contains the version of the ZigBee stack currently running on the client. This attribute is optional.
- `u32DownloadedFileVersion` contains the file version of the downloaded upgrade image on the client. This attribute is optional.
- `u16DownloadedStackVersion` contains the version of the ZigBee stack for which the downloaded upgrade image was built. This attribute is optional.
- `u8ImageUpgradeStatus` contains the status of the client device in relation to image downloads and upgrades. This attribute is mandatory and the possible values are shown in the table below.

u8ImageUpgradeStatus	Status	Notes
0x00	Normal	Has not participated in a download/upgrade or the previous download/upgrade was unsuccessful
0x01	Download in progress	Client is requesting and successfully receiving blocks of image data from server
0x02	Download complete	All image data received and image saved to memory
0x03	Waiting to upgrade	Waiting for instruction from server to upgrade from the saved image
0x04	Count down	Server instructs the Client to count down to start of upgrade
0x05	Wait for more	Client is waiting for further upgrade image(s) from server - relevant to multi-processor devices, where each processor requires a different image
0x06 - 0xFF	Reserved	-

- `u16ManfId` contains the device's manufacturer code, assigned by the ZigBee Alliance. This attribute is optional.
- `u16ImageType` contains an image type identifier for the upgrade image that is currently being downloaded to the client or waiting on the client for the upgrade process to begin. When neither of these cases apply, the attribute is set to 0xFFFF. This attribute is optional.
- `u16MinBlockRequestDelay` is the minimum time, in seconds, that the local client must wait between submitting consecutive block requests to the server during an image download. It is used by the 'rate limiting' feature to control the average download rate to the client. The attribute can take values in the range 0 to OTA_BLOCK_REQUEST_DELAY_MAX_VALUE seconds, where this upper limit can be defined in the **zcl_options.h** file (see Section 49.13) - if undefined, its default value is 5 seconds. The value 0x0000 (default) indicates that the download can be performed at the full rate with no minimum delay between block requests. This attribute is optional.
- `u16ClusterRevision` is a mandatory attribute that specifies the revision of the cluster specification on which this cluster instance is based. The cluster specification in the ZCL r6 corresponds to a cluster revision of 1. The value is incremented by one for each subsequent revision of the cluster specification. This attribute is also described in Section 2.4.

Thus, the OTA Upgrade cluster structure contains only two mandatory elements, `u64UgradeServerID` and `u8ImageUpgradeStatus`. The remaining elements are optional, each being enabled/disabled through a corresponding macro defined in the **zcl_options.h** file (see Section 49.13).

49.4 Basic Principles

Over-the-Air (OTA) Upgrade allows the application software on a ZigBee node to be upgraded with minimal disruption to node operation and without physical intervention by the user/installer. For example, there is no need for a cabled connection to the node. Using this technique, the replacement software is distributed to nodes through the wireless network, allowing application upgrades to be performed remotely.

The software upgrade is performed from a node which acts as an OTA Upgrade cluster server, which is able to obtain the upgrade software from an external source. The nodes that receive the upgrade software act as OTA Upgrade cluster clients. The server node and client node(s) may be from different manufacturers.

The download of an application image from the server to the network is done on a per client basis and follows normal network routes (including routing via Routers). This is illustrated in the figure below.

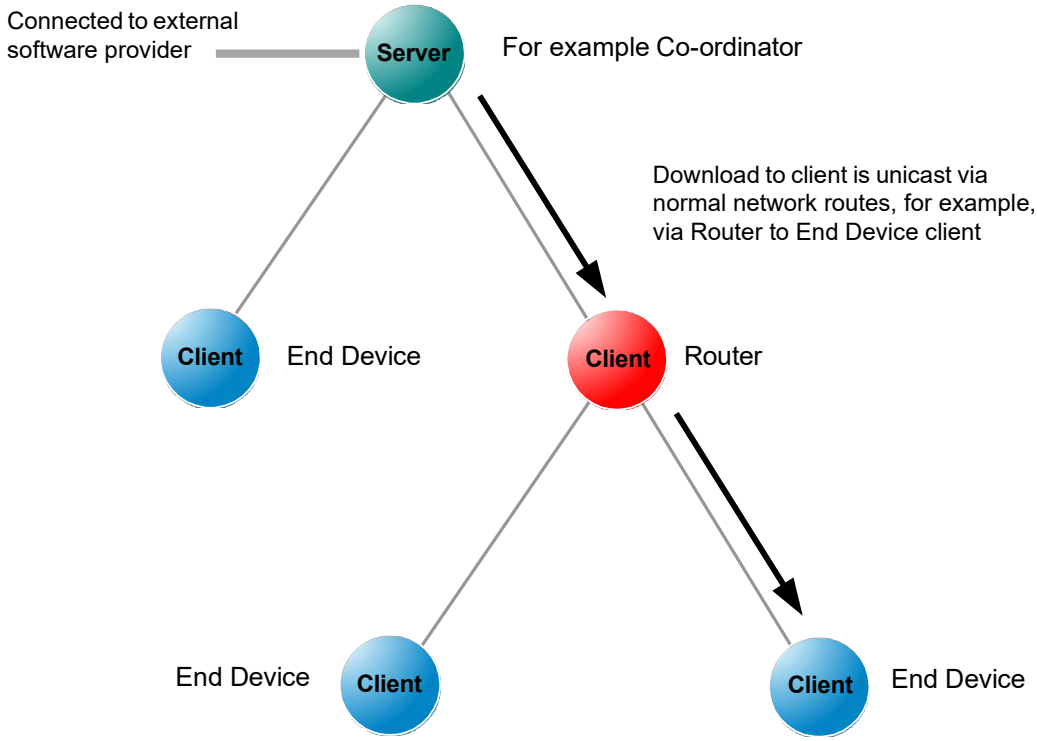


Figure 10. OTA Download Example

The upgrade application is downloaded into Flash memory internal to the device on the client node. Note that the upper section of Flash memory should normally be reserved for persistent data storage - for example, in an 8-sector Flash device, Sector 7 is used for persistent data storage, leaving Sectors 0-6 available to store application software.

The requirements of the devices which act as the OTA Upgrade cluster server and clients are detailed in the sub-sections below. See Connectivity Framework Reference Manual for details of the Non-Volatile Memory Manager (NVM).

49.4.1 OTA Upgrade Cluster Server

The OTA Upgrade cluster server is a network node that distributes application upgrades to other nodes of the network (as well as performing its own functions). The server must, therefore, be connected to the provider of the upgrade software. The server would also usually be the Coordinator of the ZigBee network.

The server may need to store different upgrade images for different nodes (possibly from different manufacturers) and must have ample Flash memory space for this purpose. Therefore, the server must keep a record of the software required by each client in the network and the software version number that the client is currently on. When a new version of an application image becomes available, the server may notify the relevant client(s) or respond to poll requests for software upgrades from the clients (see Section 49.4.2 below).

49.4.2 OTA Upgrade Cluster Client

An OTA Upgrade cluster client is a node which receives software upgrades from the server and can be any type of node in a ZigBee network. However, an End Device client which sleeps is not always available to receive notifications of software upgrades from the server and must therefore, periodically poll the server for upgrades. In fact, all types of client can poll the server, if preferred.

During a software download from server to client, the upgrade image is transferred over the air in a series of data blocks. It is the responsibility of the client (and not the server) to keep track of the blocks received and then to validate the final image. The upgrade image is initially saved to the relevant sectors of Flash memory on the client. There must be enough Flash memory space on the client to store the upgrade image and the image of the currently running software.

An OTA upgrade image is downloaded into a Flash memory of the device, utilizing Flash sectors is currently not used for the running image.

49.5 Application Requirements

In order to implement OTA upgrades, the application images for the server and clients must be designed and built according to certain requirements.

These requirements include the following:

- Inclusion of the header files **zcl_options.h** and **OTA.h**
- Inclusion of the relevant #defines in the file **zcl_options.h**, as described in Section 49.13
- Specific application initialization requirements, as outlined in Section 49.6
- Use of the Non-Volatile Memory Manager (NVM) to preserve context data, as outlined in Section 49.8.5
- Organization of Flash memory, as outlined in Section 49.8.6
- It is necessary to remove references to the Certicom security certificate, as indicated in Section 49.13

Note: *Some of above requirements differ between the server image, the first client image and client upgrade images. These differences are pointed out, where relevant, in Section 49.6 and Section 49.8.*

In addition, you should increase the CPU stack size from the default value. With OTA Upgrade, the recommended stack size is 6000 bytes. This can be done by including the following line in your application makefile:

```
__stack_size = 6000;
```

49.6 Initialization

Initialization of the various software components used with the OTA Upgrade cluster (see Section 49.5) must be performed in a particular order in the application code. The initialization could be incorporated in a function **APP_vInitialise()**, as is the case in the NXP *ZigBee PRO Application Template (JN-AN-1248)*.

Initialization must be performed in the following order:

1. The NVM module must first be initialized using the function **NvModuleInit()**.

2. The persistent data record(s) should then be initialized using the function and registered **NVM_RegisterDataSet()**.

3. The ZigBee PRO stack must now be started by first calling the function **ZPS_vSetOverrideLocalMacAddress()** to over-ride the existing MAC address, followed by **ZPS_eAplAfInit()** to initialize the Application Framework and then **ZPS_eAplZdoStartStack()** to start the stack.

4. The ZCL initialization function, **eZCL_Initialise()**, can now be called. An OTA Upgrade cluster instance should then be created using **eOTA_Create()**, followed by a call to **eOTA_UpdateClientAttributes()** or **eOTA_RestoreClientData()** on a client to initialize the cluster attributes.

5. The Flash programming of the OTA Upgrade cluster must now be initialized using the function **vOTA_FlashInit()**.

6. The required device endpoint(s) can now be registered (for example, a Simple Sensor device).

7. The function **eOTA_AllocateEndpointOTASpace()** must be called to allocate Flash memory space to an endpoint. The information provided to this function includes the numbers of the start sectors for storage of application images and the maximum number of sectors per image.

8. On the server, a set of client devices can be defined for which OTA upgrades are authorized - that is, a list of clients that are allowed to use the server for OTA upgrades. This client list is set up using the function **eOTA_SetServerAuthorisation()**.

9. For a client, a server must be found (provided this is a first-time start or a reboot with no persisted data, and so there is no record of a previous server address). This can be done by sending out a Match Descriptor Request using the function **ZPS_eAplZdpMatchDescRequest()**, described in the *ZigBee 3.0 Stack User Guide (JNUG3130)*. Once a server has been found, its address must be registered with the OTA Upgrade cluster using the function **eOTA_SetServerAddress()**.

The coding that is then required to implement OTA upgrade in the server and client applications is outlined in Section 49.7.

49.7 Implementing OTA Upgrade Mechanism

The OTA upgrade mechanism is implemented in code as described below.

Note: *The stack automatically handles part of an OTA upgrade and calls some of the OTA functions. However, if preferred, the application can handle all aspects of an OTA upgrade and filter all OTA data indications. In this case, the application must call all the relevant OTA functions (these are indicated below).*

1. On the server, when a new client image is available for download, the function **eOTA_NewImageLoaded()** should be called to request the OTA Upgrade cluster to validate the image.

   Then, optionally, the function **eOTA_SetServerParams()** can be called to set the server parameter values for the new image. Otherwise, the default parameter values will be used.

2. The server must then notify the relevant client(s) of the availability of the new image. The notification method depends on the ZigBee node type of the client:

- **Coordinator or Router client:** The server can notify the Coordinator or a Router client directly by sending an Image Notify message to the client through a call to the function **eOTA_ServerImageNotify()**. This message can be unicast, multicast or broadcast. On arrival at a client, this message will trigger an Image Notify event. If the new software is required, the client can request the upgrade image by sending a Query Next Image Request to the server through a call to **eOTA_ClientQueryNextImageRequest()**.

- **All clients:** The server cannot notify an End Device client directly, since the End Device may be asleep when a notification message is sent. Therefore, an End Device client must poll the server periodically (during wake periods) in order to establish whether new software is available. In fact, any client can implement polling of the server. The client does this by sending a Query Next Image Request to the server through a call to the function **eOTA_ClientQueryNextImageRequest()**.

  On arrival at the server, the Query Next Image Request message triggers a Query Next Image Request event.

3. The server automatically replies to the request with a Query Next Image Response (the application can also send this response by calling the function **eOTA_ServerQueryNextImageResponse()**). The contents of this response message depend on whether the client is using notifications or polling:

- **Coordinator or Router client (notifications):** The response contains details of the upgrade image, such as manufacturer, image type, image size, and file version.
- **All clients (polling):** If upgrade software is available, the response reports success and the message contains details of the upgrade image, as indicated above. If no upgrade software is available, the response simply reports failure (the client must then poll again later).

  On arrival at the client, the Query Next Image Response message triggers a Query Next Image Response event.

4. The OTA Upgrade cluster on the client now automatically requests the upgrade image one block at a time by sending an Image Block Request to the server (this request can also be sent by the application through a call to the function **eOTA_ClientImageBlockRequest()**). The maximum size of a block and the time interval between requests can both be configured in the header file **zcl-_options.h** - see Section 49.13.

On arrival at the server, the Image Block Request message triggers an Image Block Request event.

5. The server automatically responds to each block request with an Image Block Response containing a block of data (the application can also send this response by calling the function **eOTA_ServerImageBlockResponse()**).

On arrival at the client, the Image Block Response message triggers an Image Block Response event.

6. The client determines when the entire image has been received (by referring to the image size that was quoted in the Query Next Image Response before the download started). Once the final block of image data has been received, the client application should transmit an Upgrade End Request to the server (that is, by calling **eOTA_HandleImageVerification()**).

This Upgrade End Request may report success or an invalid image. In the case of an invalid image, the image will be discarded by the client, which may initiate a new download of the image by sending a Query Next Image Request to the server.

On arrival at the server, the Upgrade End Request message triggers an Upgrade End Request event.

Note: *An Upgrade End Request may also be sent to the server during a download in order to abort the download.*

7. The server replies to the request with an Upgrade End Response containing an instruction of when the client should use the downloaded image to upgrade the running software on the node (the message contains both the current time and the upgrade time, and hence an implied delay).

On arrival at the client, the Upgrade End Response message triggers an Upgrade End Response event.

8. The client will then count down to the upgrade time (in the Upgrade End Response) and on reaching it, start the upgrade. If the upgrade time has been set to an indefinite value (represented by 0xFFFFFFFF), the client should poll the server for an Upgrade Command at least once per minute and start the upgrade once this command has been received.

9. Once triggered on the client, the upgrade process proceeds as follows (although the details will be manufacturer-specific):

a) A reboot of the device is initiated causing the default bootloader to run.

b) The bootloader scans through Flash looking for various image headers, including the running one and the one received via OTA. If the OTA image in newer, indicated by version number in the header, this is selected as the new running image.

Note: *The client automatically invalidates the existing image and validates the new upgrade image once the allotted upgrade time is reached.*

c) The new software image is then executed.

Query Jitter

The 'query jitter' mechanism can be used to prevent a flood of replies to an Image Notify broadcast or multicast (Step 2 above). The server includes a number, n, in the range 1-100 in the notification. If interested in the image, the receiving client generates a random number in the range 1-100. If this number is greater than n, the client discards the notification, otherwise it responds with a Query Next Image Request. This results in only a fraction of interested clients responding to each broadcast/multicast and therefore helps to avoid traffic congestion.

49.8 Ancillary Features and Resources for OTA Upgrade

As indicated in Section 49.5, in order to implement OTA upgrades, a number of other software features and resources are available. These are described in the sub-sections below.

49.8.1 Rate Limiting

During busy periods when the OTA Upgrade server is downloading images to multiple clients, it is possible to prevent OTA traffic congestion by limiting the download rates to individual clients. This is achieved by introducing a minimum time-delay between consecutive Image Block Requests from a client - for example, if this delay is set to 500 ms for a particular client then after sending one block request to the server, the client must wait at least 500 ms before sending the next block request. This has the effect of restricting the average OTA download rate from the server to the client.

This ‘block request delay’ can be set to different values for different clients. This allows OTA downloads to be prioritized by granting more download bandwidth to some clients than to others. This delay for an individual client can also be modified by the server during a download, allowing the server to react in real-time to varying OTA traffic levels.

The implementation of the above rate limiting is described below and is illustrated in Figure 11.

‘Block Request Delay’ Attribute

The download rate to an individual client is controlled using the optional attribute `u16MinBlockRequestDelay` of the OTA Upgrade cluster (see Section 49.3) on the client. This attribute contains the ‘block request delay’ for the client (described above), in milliseconds, and must be enabled on the client only (see below).

Note: *The `u16MinBlockRequestDelay` attribute is the minimum time-interval between block requests. The application on the client can implement longer intervals between these requests (a slower download rate), if required.*

Enabling the Rate Limiting Feature

In order to use the rate limiting feature during an OTA upgrade, the macro OTA_CLD_ATTR_REQUEST_DELAY must be defined in the **zcl_options.h** file for both the participating client(s). This enables the `u16MinBlockRequestDelay` attribute in the OTA Upgrade cluster structure.

Implementation in the Server Application

The application on the OTA Upgrade server device can control the OTA download rate to an individual client by remotely setting the value of the 'block request delay' attribute on the client. However, first the server must determine whether the client supports the rate limiting feature. The server can do this in either of two ways:

- It can attempt to read the `u16MinBlockRequestDelay` attribute in the OTA Upgrade cluster on the client - if rate limiting is not enabled on the client, this read will yield an error.
- It can check whether the first Image Block Request received from the client contains a 'block request delay' field - if present, this value is passed to the application in the event E_CLD_OTA_COMMAND_BLOCK_REQUEST.

The server can change the value of the 'block request delay' attribute on the client at any time, even during a download. To do this, the server includes the new attribute value in an Image Block Response with status OTA_STATUS_WAIT_FOR_DATA. This is achieved in the application code through a call to the function **eOTA_SetWaitForDataParams()** following an Image Block Request (indicated by an E_CLD_OTA_COMMAND_BLOCK_REQUEST event). The new attribute value specified in this function call is included in the subsequent Image Block Response and is automatically written to the OTA Upgrade cluster on the client.

The server may update the 'block request delay' attribute on a client multiple times during a download in order to react to changing OTA traffic conditions. If the server is downloading an image to only one client then it may choose to allow this download to proceed at the full rate (specified by a zero value of the attribute on the client). However, if two or more clients request downloads at the same time, the server may choose to limit their download rates (by setting the attribute to non-zero values on the clients). The download to one client can be given higher priority than other downloads by setting the attribute on this client to a lower value.

Implementation in the Client Application

The application on the OTA Upgrade client device must control a millisecond timer (a timer with a resolution of one millisecond) to support rate limiting. This timer is used to time the delay between receiving an Image Block Response and submitting the next Image Block Request.

During an image download, a received Image Block Response with the status OTA_STATUS_WAIT_FOR_DATA may contain a new value for the 'block request delay' attribute (this type of response may arrive at the start of a download or at any time during the download). The client will automatically write this new value to the `u16MinBlockRequestDelay` attribute in the local OTA Upgrade cluster structure and will also generate the event E_ZCL_CBET_ENABLE_MS_TIMER (provided that the new attribute value is non-zero).

The E_ZCL_CBET_ENABLE_MS_TIMER event prompts the application to start the millisecond timer for a timed interval greater than or equal to the new value of the 'block request delay' attribute. The application can obtain this new attribute value (in milliseconds) from the event via:

```
sZCL_CallBackEvent.uMessage.u32TimerPeriodMs
```

The millisecond timer is started for a particular timed interval. The expiry of this timer is indicated by an E_ZCL_CBET_TIMER_MS event, which is handled as described in Section 3.2. The client will then send the next Image Block Request.

After sending an Image Block Request:

- If the client now generates an E_ZCL_CBET_DISABLE_MS_TIMER event, this indicates that the last of the Image Block Request (for the required image) has been sent and the application should disable the millisecond timer.
- Otherwise, the application must start the next timed interval (until the next request).

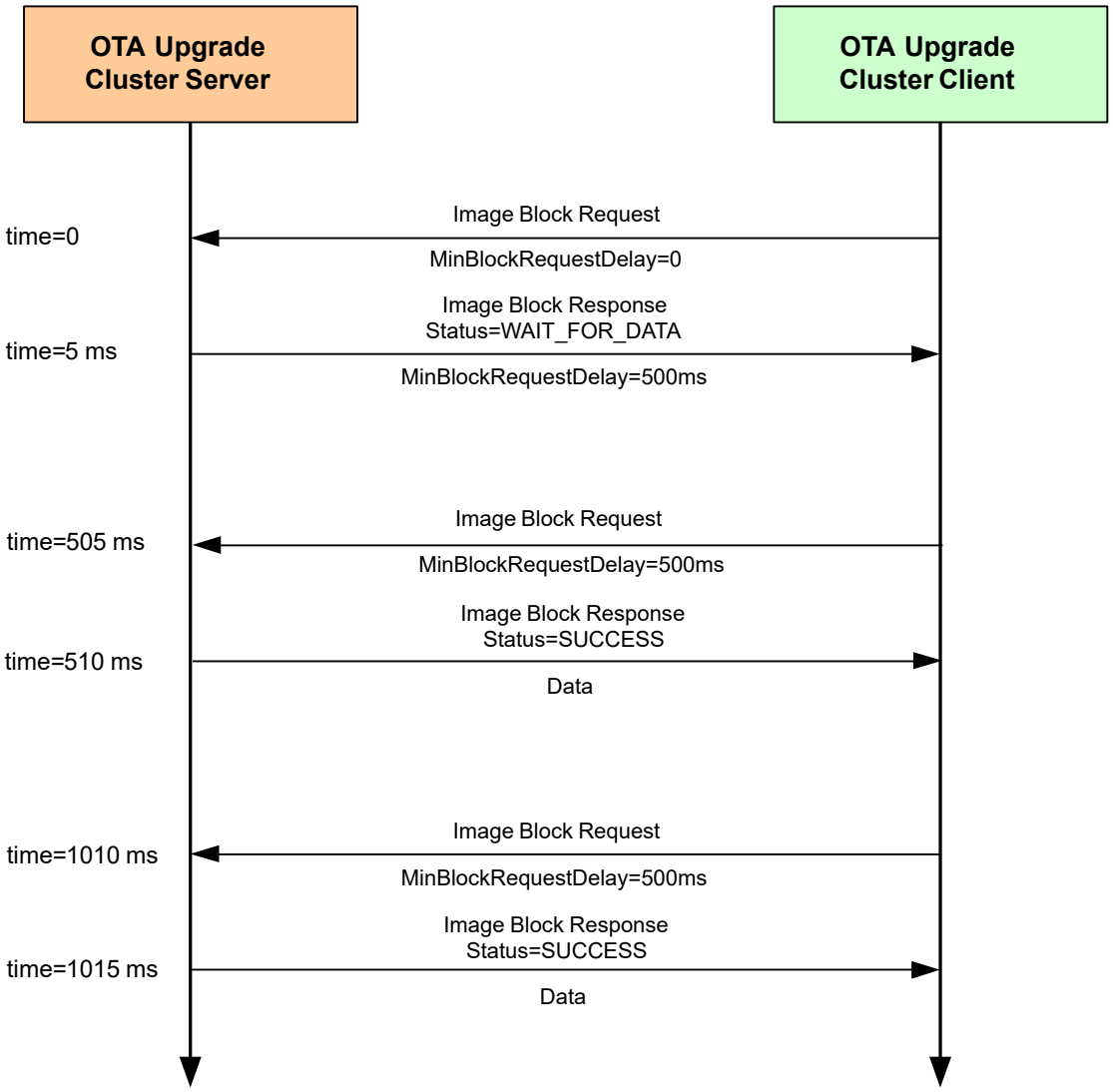


Figure 11. Example of Rate limiting exchange

49.8.2 Device-Specific File Downloads

An OTA Ugrade client can request a file (from the server) that is specific to the client device. This file may contain non-firmware data such as security credentials, configuration data or log data. The process of making this request and receiving the file is described in the table below for both the client and server sides.

	On Client	On Server
1	Client application sends a Query Specific File Request to the server through a call to **eOTA_ClientQuerySpecific FileRequest()**.	
2		On arrival at the server, the Query Specific File Request triggers the event E_CLD_OTA_COMMAND_QUERY_ SPECI-FIC_FILE_REQUEST.
3		Server automatically replies to the request with a Query Specific File Response - the application can also send

	On Client	On Server
		a response using **eOTA_ServerQuerySpecificFile Response()**.
4	On arrival at the client, the Query Specific File Response triggers the event E_CLD_OTA_COMMAND_QUERY_ SPECIFIC_FILE_RESPONSE.	
5	Client obtains status from Query Specific File Response. If status is SUCCESS, the client automatically requests the device-specific file one block at a time by sending Image Block Requests to the server.	
6		On arrival at the server, each Image Block Request triggers an Image Block Request event.
7		Server automatically responds to each block request with an Image Block Response containing a block of device-specific file data.
8	After receiving each Image Block Response, the client generates the event E_CLD_OTA_INTERNAL_ COMMAND_SPECIFIC_FILE_BLOCK_RESPONSE.	
9	A callback function is invoked on the client to handle the event and store the data block (it is the responsibility of the application to store the data in a convenient place).	
10	Client determines when the entire file has been received (by referring to the file size that was quoted in the Query Specific File Response before the download started). Once all the file blocks have been received: • E_CLD_OTA_INTERNAL_COMMAND_SPECIFIC_FIL E_DL_COMPLETE event is generated by the client to indicate that the file transfer is complete. • The file can optionally be verified by application. • Client sends an Upgrade End Request to the server to indicate that the download is complete, where this request is the result of an application call to the function **eOTA_SpecificFileUpgradeEndRequest()**.	
11		On arrival at the server, the Upgrade End Request triggers an Upgrade End Request event.
12		Server may reply to the Upgrade End Request with an Upgrade End Response containing an instruction of when the client should use the device-specific file (the message contains both the current time and the upgrade time, and hence an implied delay) - see Footnotes 1 and 2 below.
13	On arrival at the client, the Upgrade End Response triggers an Upgrade End Response event - see Footnotes 1 and 2 below.	
14	Client will then count down to the upgrade time (in the Upgrade End Response) and, on reaching it, will generate the event E_CLD_OTA_INTERNAL_ COMMAND_SPECIFIC_FILE_USE_NEW_FILE. Finally, it is the responsibility of the application to use device-specific file as appropriate.	

Footnotes

1. For a device-specific file download, it is not mandatory for the server to send an Upgrade End Response to the client. In the case of a client which has just finished retrieving a log file from the server, the Upgrade End Response may not be needed. However, if the client has just retrieved a file containing security credentials or configuration data, the Upgrade End Response may be needed to notify the client of when to apply the file. The decision of whether to send an Upgrade End Response for a device-specific file download is manufacturer-specific.

2. If an Upgrade End Response is not received from the server, the client will perform 3 retries to get the response. If it still does not receive a response, the client will generate the event E_CLD_OTA_INTERNAL_COMMAND_SPECIFIC_FILE_NO_UPGRADE_END_RESPONSE.

49.8.3 Image Block Size and Fragmentation

An OTA Upgrade image is normally requested by the OTA Upgrade client one block at a time. The ZigBee frame for the OTA transfer contains various header data as well as payload data and, for this reason, the payload data is limited to about 48 bytes. Therefore, to transfer one image block per frame, the block size must be restricted to 48 bytes or less. The maximum block size can be configured at compile-time through the OTA_MAX_BLOCK_SIZE define in the **zcl_options.h** file (see Section 49.13).

A block size of greater than 48 bytes can be used but the image block will need to be transferred across two or more ZigBee frames. In this case, fragmentation must be enabled in which the image block data that is assembled in an APDU (Application Protocol Data Unit) on the server is fragmented into multiple NPDUs (Network Protocol Data Unit) for OTA transfer, where one NPDU is transferred in a single ZigBee frame. Fragmentation is enabled on the OTA Upgrade server and client using network parameters of the ZigBee PRO stack, as follows:

- **On the server:** Set the parameter *Maximum Number of Transmitted Simultaneous Fragmented Messages* to a non-zero value to allow transmitted messages to be fragmented.
- **On the client:** Set the parameter *Maximum Number of Received Simultaneous Fragmented Messages* to a non-zero value to allow received fragmented messages to be re-assembled.

The network parameter values are set using the ZPS Configuration Editor and are described in the *ZigBee 3.0 Stack User Guide (JNUG3130)*.

Note: ***Note:*** *The 48-byte limit on the payload data in a ZigBee frame is also applicable when image data is requested and transferred one page at a time (see Section 49.8.4). In this case, fragmentation may need to be enabled.*

The maximum APDU size must always be greater than the size of an Image Block Response. It is set through the APDU *Size* parameter of the PDU Manager, where this parameter is amongst the Advanced Device Parameters that can be configured using the ZPS Configuration Editor.

Depending on the image block size, fragmentation is not always an efficient way of transferring image blocks, as the payload of the final NPDU fragment may contain little data and be mostly empty. For example, if the image block size is set to 64 bytes and fragmentation is enabled, each block is transferred in two ZigBee frames, the first may contain 48 bytes of data and the second may contain only 16 bytes of data, leaving 32 empty bytes in the payload. In contrast, if the block size is set to 48 bytes without fragmentation, two consecutive frames would carry 96 bytes of data, and the image transfer would require fewer frames. This is particularly important when transferring an application image to a battery-powered End Device that needs to conserve energy.

49.8.4 Page Requests

An OTA Upgrade client normally requests image data from the server one block at a time, by sending an Image Block Request when it is ready for the next block. The number of requests can be reduced by requesting the image data one page at a time, where a page may contain many blocks of data. Requesting data by pages reduces the OTA traffic and, in the case of battery-powered client device, extends battery life.

A page of data is requested by sending an Image Page Request to the server. This request contains a page size, which indicates the number of data bytes that should be returned by the server following the request (and before the next request is sent, if any). The server still sends the data one block at a time in Image Block Responses. The Image Page Request also specifies the maximum number of bytes that the client device can receive in any one OTA message and the block size must therefore not exceed this limit (in general, the page size should be a multiple of this limit).

It is the responsibility of the client to keep track of the amount of data so far received since the last Image Page Request was issued - this count is updated after each Image Block Response received. Once this count reaches the page size in the request, the client will issue the next Image Page Request (if the download is not yet complete).

During a download that uses page requests:

- If the client fails to receive one or more of the requested blocks then the next Image Page Request will request data starting from the offset which corresponds to the first missing block.
- If the client fails to receive all the blocks requested in an Image Page Request then the same request will be repeated up to two more times - if the requested data still fails to arrive, the client will switch to using Image Block Requests to download the remaining image data.

An Image Page Request also contains a ‘response spacing’ value. This indicates the minimum time-interval, in milliseconds, that the server should insert between consecutive Image Block Responses. If the client is a sleepy End Device, it may specify a long response spacing so that it can sleep between consecutive Image Block Responses, or it may specify a short response spacing so that it can quickly receive all blocks requested in a page and sleep between consecutive Image Page Requests.

The implementation of the above page requests in an application is described below. The OTA image download process using page requests is similar to the one described in Section 49.7, except the client submits Image Page Requests to the server instead of Image Block Requests.

Enabling the Page Requests Feature

In order to use page requests, the macro OTA_PAGE_REQUEST_SUPPORT must be defined in the **zcl_options.h** file for the server and client.

In addition, values for the page size and response spacing can also be defined in this file for the client (if non-default values are required) - see below and Section 49.13.

Implementation in the Server Application

The application on the OTA Upgrade server device must control a millisecond timer (a timer with a resolution of one millisecond) to support page requests. This timer is used to implement the ‘response spacing’ specified in an Image Page Request - that is, to time the interval between the transmissions of consecutive Image Block Responses (sent out in response to the Image Page Request).

When the server receives an Image Page Request, it will generate the event E_ZCL_CBET_ENABLE_MS_TIMER to prompt the application to start the millisecond timer for a timed interval equal in value to the ‘response spacing’ in the request. The application can obtain this value (in milliseconds) from the event via:

```
sZCL_CallBackEvent.uMessage.u32TimerPeriodMs
```

The millisecond timer is started for a particular timed interval. The expiry of this timer is indicated by an E_ZCL_CBET_TIMER_MS event, which is handled as described in Section 3.2. The server will then send the next Image Block Response.

After sending an Image Block Response:

- If the server now generates an E_ZCL_CBET_DISABLE_MS_TIMER event, this indicates that the last of the Image Block Responses (for the Image Page Request) has been sent and the application should disable the millisecond timer.
- Otherwise, the application must start the next timed interval (until the next response).

Implementation in the Client Application

There is nothing specific to do in the client application to implement page requests. Provided that page requests have been enabled in the **zcl_options.h** file for the client (see above), page requests will be automatically implemented by the stack instead of block requests for OTA image downloads. The page size (in bytes) and response spacing (in milliseconds) for these requests can be specified through the following macros in the **zcl_options.h** file (see Section 49.13):

- OTA_PAGE_REQ_PAGE_SIZE
- OTA_PAGE_REQ_RESPONSE_SPACING

The default values are 512 bytes and 300 ms, respectively.

However, the client application can itself submit an Image Page Request to the server by calling the function **eOTA_ClientImagePageRequest()**. In this case, the page size and response spacing are specified in the Image Page Request payload structure as part of this function call.

The client handles the resulting Image Block Responses as described in Section 49.7 for standard OTA downloads.

49.8.5 Persistent Data Management

The OTA Upgrade cluster on a client requires context data to be preserved in non-volatile memory to facilitate a recovery of the OTA Upgrade status following a device reboot. The Non-Volatile Memory Manager (NVM) module should be used to perform this data saving and recovery. The NVM module is implemented as described in the *Connectivity Framework Reference Manual*.

Persistent data is normally be stored in the upper sector of the devices Flash memory. Thus, when the NVM module is initialized, these sectors should be specified (just these sectors should be managed by the NVM module).

When it needs to save context data, the OTA Upgrade cluster will generate the event E_CLD_OTA_INTERNAL_COMMAND_SAVE_CONTEXT, which will also contain the data to be saved. A user-defined callback function can then be invoked to perform the data storage using functions of the NVM module.

The OTA Upgrade cluster is implemented for an individual application/endpoint. Therefore, the NVM module should also be implemented per endpoint. The following code illustrates the reservation of memory space for persistent data per endpoint.

```
typedef struct
{
            uint8 u8Endpoints[APP_NUM_OF_ENDPOINTS];
            uint8 eState; // Current application state to re-instate
    tsOTA_PersistedData sPersistedData[APP_NUM_OF_ENDPOINTS];
} tsDevice;
PUBLIC tsDevice s_sDevice;
```

If a client is restarted and persisted data is available on the device, the OTA Upgrade cluster data should be restored using the function **eOTA_RestoreClientData()**.

49.8.6 Flash Memory Organization

Flash memory should be organized such that the application images are stored from Sector 0 and, if required, persistent data is stored in the final sectors.

The storage of applications and persistent data in Flash memory is described further below. Guidance on the organization of OTA upgrade applications in the devices internal Flash memory is also provided Appendix.

Application Images

As part of application initialization (see Section 49.6), the OTA Upgrade cluster must be informed of the storage arrangements for application images in Flash memory. This is done through the function **eOTA_AllocateEndpointOTASpace()**, which applies to a specified endpoint (normally the endpoint of the application which calls the function). The information provided via this function includes:

- Start sector for each image that can be stored (specified through an array with one element per image). s
- Number of images for the endpoint (the maximum number of images per endpoint is specified in the **zcl_options.h** file - see Section 49.13)
- Maximum number of sectors per image
- Type of node (server or client)
- Public key for signed images

Persistent Data

The storage of persistent data is handled by the NVM module (see Section 49.8.5) and the sector used is specified as part of the NVM initialization through **NvModuleInit()** - the final sector of Flash memory should be specified.

49.8.7 Low-Voltage Flag

An OTA Upgrade cluster client should not attempt to participate in an OTA upgrade if the supply voltage to the host hardware device is low (below the normal operating voltage for the device). On the device, sufficient voltage is required to write to the internal Flash There may be a number of reasons for a sudden drop in supply voltage - for example, the voltage on a battery-powered node may fall when the battery is near the end of its life.

The OTA Upgrade cluster incorporates a mechanism which, if enabled, stops the cluster client from sending Image Block Requests to the server when the local supply voltage becomes low. This mechanism allows the application to set a low-voltage flag which, when set, automatically suspends the block requests. When the flag is cleared, the block requests are automatically resumed.

If required, use of the low-voltage flag and associated mechanism must be enabled at compile-time by including the following line in the **zcl_options.h** file:

```
#define OTA_UPGRADE_VOLTAGE_CHECK
```

It is the responsibility of the application to check the supply voltage. This check is system-specific and may be performed periodically or using a voltage monitoring feature - for example, on the device, the Supply Voltage Monitor (SVM) can be used, which is described in the MCUXpresso SDK API Reference Manual.

The application can use the function **vOTA_SetLowVoltageFlag()** to configure the low-voltage flag. This function is detailed in Section 49.10.3.

When a low voltage is detected, the application should make the following function call to set the low-voltage flag and suspend Image Block Requests:

```
vOTA_SetLowVoltageFlag(TRUE);
```

When the voltage is restored to a normal level, the application should make the following function call to clear the low-voltage flag and resume Image Block Requests:

```
vOTA_SetLowVoltageFlag(FALSE);
```

49.9 OTA Upgrade events

The events that can be generated on an OTA Upgrade cluster server or client are defined in the structure `teOTA_UpgradeClusterEvents` (see Section 49.12.2). The events are listed in the table below, which also indicates on which side of the cluster (server or client) the events can occur:

Table 117. OTA Upgrade Events

Cluster Side(s)	Event
Server	E_CLD_OTA_COMMAND_QUERY_NEXT_IMAGE_REQUEST
	E_CLD_OTA_COMMAND_BLOCK_REQUEST
	E_CLD_OTA_COMMAND_PAGE_REQUEST
	E_CLD_OTA_COMMAND_UPGRADE_END_REQUEST
	E_CLD_OTA_COMMAND_QUERY_SPECIFIC_FILE_REQUEST
	E_CLD_OTA_INTERNAL_COMMAND_SEND_UPGRADE_END_RESPONSE
	E_CLD_OTA_INTERNAL_COMMAND_CO_PROCESSOR_IMAGE_BLOCK_REQUEST
Client	E_CLD_OTA_COMMAND_IMAGE_NOTIFY
	E_CLD_OTA_COMMAND_QUERY_NEXT_IMAGE_RESPONSE
	E_CLD_OTA_COMMAND_BLOCK_RESPONSE
	E_CLD_OTA_COMMAND_UPGRADE_END_RESPONSE
	E_CLD_OTA_COMMAND_QUERY_SPECIFIC_FILE_RESPONSE
	E_CLD_OTA_INTERNAL_COMMAND_TIMER_EXPIRED
	E_CLD_OTA_INTERNAL_COMMAND_POLL_REQUIRED
	E_CLD_OTA_INTERNAL_COMMAND_RESET_TO_UPGRADE
	E_CLD_OTA_INTERNAL_COMMAND_SAVE_CONTEXT
	E_CLD_OTA_INTERNAL_COMMAND_OTA_DL_ABORTED
	E_CLD_OTA_INTERNAL_COMMAND_CO_PROCESSOR_BLOCK_RESPONSE
	E_CLD_OTA_INTERNAL_COMMAND_CO_PROCESSOR_DL_ABORT
	E_CLD_OTA_INTERNAL_COMMAND_CO_PROCESSOR_IMAGE_DL_COMPLETE
	E_CLD_OTA_INTERNAL_COMMAND_CO_PROCESSOR_SWITCH_TO_NEW_IMAGE
	E_CLD_OTA_INTERNAL_COMMAND_SPECIFIC_FILE_BLOCK_RESPONSE
	E_CLD_OTA_INTERNAL_COMMAND_SPECIFIC_FILE_DL_COMPLETE
	E_CLD_OTA_INTERNAL_COMMAND_SPECIFIC_FILE_DL_ABORT
	E_CLD_OTA_INTERNAL_COMMAND_SPECIFIC_FILE_USE_NEW_FILE

Table 117. OTA Upgrade Events*...continued*

Cluster Side(s)	Event
	E_CLD_OTA_INTERNAL_COMMAND_SPECIFIC_FILE_NO_UPGRADE_END_RESPONSE
	E_CLD_OTA_COMMAND_QUERY_NEXT_IMAGE_RESPONSE_ERROR
	E_CLD_OTA_INTERNAL_COMMAND_VERIFY_SIGNER_ADDRESS
	E_CLD_OTA_INTERNAL_COMMAND_RCVD_DEFAULT_RESPONSE
	E_CLD_OTA_INTERNAL_COMMAND_VERIFY_IMAGE_VERSION
	E_CLD_OTA_INTERNAL_COMMAND_SWITCH_TO_UPGRADE_DOWNGRADE
	E_CLD_OTA_INTERNAL_COMMAND_REQUEST_QUERY_NEXT_IMAGES
	E_CLD_OTA_INTERNAL_COMMAND_OTA_START_IMAGE_VERIFICATION_IN_LOW_PRIORITY
	E_CLD_OTA_INTERNAL_COMMAND_FAILED_VALIDATING_UPGRADE_IMAGE
	E_CLD_OTA_INTERNAL_COMMAND_FAILED_COPYING_SERIALIZATION_DATA
	E_CLD_OTA_BLOCK_RESPONSE_TAG_OTHER_THAN_UPGRADE_IMAGE
Both	E_CLD_OTA_INTERNAL_COMMAND_LOCK_FLASH_MUTEX
	E_CLD_OTA_INTERNAL_COMMAND_FREE_FLASH_MUTEX

OTA Upgrade events are treated as ZCL events. Thus, an event is received by the application, which wraps the event in a `tsZCL_CallBackEvent` structure and passes it into the ZCL using the function **vZCL_EventHandler()** - for further details of ZCL event processing, refer to Chapter 3.

The above events are outlined in the sub-sections below.

49.9.1 Server-side Events

- **E_CLD_OTA_COMMAND_QUERY_NEXT_IMAGE_REQUEST**
  This event is generated on the server when a Query Next Image Request is received from a client to enquire whether a new application image is available for download. The event may result from a poll request from the client or may be a consequence of an Image Notify message previously sent by the server. The server reacts to this event by returning a Query Next Image Response.
- **E_CLD_OTA_COMMAND_BLOCK_REQUEST**
  This event is generated on the server when an Image Block Request is received from a client to request a block of image data as part of a download. The application reacts to this event by returning an Image Block Response containing a data block.
- **E_CLD_OTA_COMMAND_PAGE_REQUEST**
  This event is generated on the server when an Image Page Request is received from a client to request a page of image data as part of a download.
- **E_CLD_OTA_COMMAND_UPGRADE_END_REQUEST**
  This event is generated on the server when an Upgrade End Request is received from a client to indicate that the complete image has been downloaded and verified. The application reacts to this event by returning an Upgrade End Response.
- **E_CLD_OTA_COMMAND_QUERY_SPECIFIC_FILE_REQUEST**
  This event is generated on the server when a Query Specific File Request is received from a client to request a particular application image. The server reacts to this event by returning a Query Specific File Response.
- **E_CLD_OTA_INTERNAL_COMMAND_SEND_UPGRADE_END_RESPONSE**
  This event is generated on the server to notify the application that the stack is going to send an Upgrade End Response to a client. No specific action is required by the application on the server.

49.9.2 Client-side Events

- **E_CLD_OTA_COMMAND_IMAGE_NOTIFY**
  This event is generated on the client when an Image Notify message is received from the server to indicate that a new application image is available for download. If the client decides to download the image, the application should react to this event by sending a Query Next Image Request to the server using the function **eOTA_ClientQueryNextImageRequest()**.
- **E_CLD_OTA_COMMAND_QUERY_NEXT_IMAGE_RESPONSE**
  This event is generated on the client when a Query Next Image Response is received from the server (in response to a Query Next Image Request) to indicate whether a new application image is available for download. If a suitable image is reported, the client initiates a download by sending an Image Block Request to the server.
- **E_CLD_OTA_COMMAND_BLOCK_RESPONSE**
  This event is generated on the client when an Image Block Response is received from the server (in response to an Image Block Request) and contains a block of image data which is part of a download. Following this event, the client can request the next block of image data by sending an Image Block Request to the server or, if the entire image has been received and verified, the client can close the download by sending an Upgrade End Request to the server.
- **E_CLD_OTA_COMMAND_UPGRADE_END_RESPONSE**
  This event is generated on the client when an Upgrade End Response is received from the server (in response to an Upgrade End Request) to confirm the end of a download. This event contains the time delay before the upgrade of the running application must be performed.
- **E_CLD_OTA_COMMAND_QUERY_SPECIFIC_FILE_RESPONSE**
  This event is generated on the client when a Query Specific File Response is received from the server (in response to a Query Specific File Request) to indicate whether the requested application image is available for download.
- **E_CLD_OTA_INTERNAL_COMMAND_TIMER_EXPIRED**
  This event is generated on the client when the local one-second timer has expired. It is an internal event and is not passed to the application.
- **E_CLD_OTA_INTERNAL_COMMAND_POLL_REQUIRED**
  This event is generated on the client to prompt the application to poll the server for a new application image by calling the function **eOTA_ClientQueryNextImageRequest()**.
- **E_CLD_OTA_INTERNAL_COMMAND_RESET_TO_UPGRADE**
  This event is generated on the client to notify the application that the stack is going to reset the device. No specific action is required by the application.
- **E_CLD_OTA_INTERNAL_COMMAND_SAVE_CONTEXT**
  This event prompts the client application to store context data in Flash memory. The data to be stored is passed to the application within this event.
- **E_CLD_OTA_INTERNAL_COMMAND_OTA_DL_ABORTED**
  This event is generated on a client if the received image is invalid or the client has aborted the image download. This allows the application to request the new image again.
- **E_CLD_OTA_INTERNAL_COMMAND_SPECIFIC_FILE_BLOCK_RESPONSE**
  This event is generated on the client when an Image Block Response is received from the server in response to an Image Block Request for a device-specific file. The event contains a block of file data which is part of a download. Following this event, the client stores the data block in an appropriate location and can request the next block of file data by sending an Image Block Request to the server (if the complete image has not yet been received and verified).
- **E_CLD_OTA_INTERNAL_COMMAND_SPECIFIC_FILE_DL_COMPLETE**

This event is generated on the client when the final Image Block Response of a device-specific file download has been received from the server - the event indicates that all the data blocks that make up the file have been received.

- **E_CLD_OTA_INTERNAL_COMMAND_SPECIFIC_FILE_USE_NEW_FILE**
  This event is generated on the client following a device-specific file download to indicate that the file can now be used by the client. At the end of the download, the server sends an Upgrade End Response that may include an 'upgrade time' - this is the UTC time at which the new file can be applied. Thus, on receiving this response, the client starts a timer and, on reaching the upgrade time, generates this event.
- **E_CLD_OTA_INTERNAL_COMMAND_SPECIFIC_FILE_DL_ABORT**
  This event is generated to indicate that the OTA Upgrade cluster needs to abort a device-specific file download. Following this event, the application should discard data that has already been received as part of the aborted download.
- **E_CLD_OTA_INTERNAL_COMMAND_SPECIFIC_FILE_NO_UPGRADE_END_RESPONSE**
  This event is generated when no Upgrade End Response has been received for a device-specific file download. The client makes three attempts to obtain an Upgrade End Response. If no response is received, the client raises this event.

Note: *For a device-specific file download, it is not mandatory for the server to send an Upgrade End Response. The decision of whether to send the Upgrade End Response is manufacturer-specific.*

- **E_CLD_OTA_COMMAND_QUERY_NEXT_IMAGE_RESPONSE_ERROR**
  This event is generated on the client when a Query Next Image Response message is received from the server, in response to a Query Next Image Request with a status of Invalid Image Size.
- **E_CLD_OTA_INTERNAL_COMMAND_RCVD_DEFAULT_RESPONSE**
  This event is generated on the client when a default response message is received from the server, in response to a Query Next Image Request, Image Block Request or Upgrade End Request. This is an internal ZCL event that results in an OTA download being aborted, thus activating the callback function for the E_CLD_OTA_INTERNAL_COMMAND_OTA_DL_ABORTED event.
- **E_CLD_OTA_INTERNAL_COMMAND_VERIFY_IMAGE_VERSION**
  This event is generated to prompt the application to verify the image version received in a Query Next Image Response. This event allows the application to verify that the new upgrade image has a valid image version. After checking the image version, the application should set the status field of the event to E_ZCL_SUCCESS (valid version) or E_ZCL_FAIL (invalid version).
- **E_CLD_OTA_INTERNAL_COMMAND_SWITCH_TO_UPGRADE_DOWNGRADE**
  This event is generated to prompt the application to verify the image version received in an upgrade end response. This event allows the application to verify that the new upgrade image has a valid image version.
  After checking the image version, the application should set the status field of the event to E_ZCL_SUCCESS (valid version) or E_ZCL_FAIL (invalid version).
- **E_CLD_OTA_INTERNAL_COMMAND_FAILED_VALIDATING_UPGRADE_IMAGE**
  This event is generated on the client when the validation of a new upgrade image fails. This validation takes place when the upgrade time is reached.
- **E_CLD_OTA_INTERNAL_COMMAND_FAILED_COPYING_SERIALIZATION_DATA**
  This event is generated on the client when the copying of serialisation data from the active image to the new upgrade image fails. This process takes place after image validation (if applicable) is completed successfully.
- **E_CLD_OTA_BLOCK_RESPONSE_TAG_OTHER_THAN_UPGRADE_IMAGE**

This event is generated on the client when an Image Block Response is received from the server but the response contains a block of data that is not upgrade image data (it may contain tags such as an integrity code or ECDA signature).Thus, this event can help the application to process tags or data other than upgrade image data.

49.9.3 Server-side and Client-side Events

- **E_CLD_OTA_INTERNAL_COMMAND_LOCK_FLASH_MUTEX**
  This event prompts the application to lock the mutex used for accesses to external Flash memory (via the SPI bus).
- **E_CLD_OTA_INTERNAL_COMMAND_FREE_FLASH_MUTEX**
  This event prompts the application to unlock the mutex used for accesses to external Flash memory (via the SPI bus).

49.10 Functions

The OTA Upgrade cluster functions that are provided in the NXP implementation of the ZCL are divided into the following three categories:

- General functions (used on server and client) - see Section 49.10.1
- Server functions - see Section 49.10.2
- Client functions - see Section 49.10.3

49.10.1 General Functions

The following OTA Upgrade cluster functions can be used on the cluster server and the cluster client:

1. eOTA_Create
2. vOTA_FlashInit
3. eOTA_AllocateEndpointOTASpace
4. vOTA_GenerateHash
5. eOTA_GetCurrentOtaHeader

49.10.1.1 eOTA_Create

```
teZCL_Status eOTA_Create(
        tsZCL_ClusterInstance *psClusterInstance,
        bool_t bIsServer,
        tsZCL_ClusterDefinition *psClusterDefinition,
        void *pvEndPointSharedStructPtr,
        uint8 u8Endpoint,
        uint8 *pu8AttributeControlBits,
        tsOTA_Common *psCustomDataStruct);
```

Description

This function creates an instance of the OTA Upgrade cluster on the specified endpoint. The cluster instance can act as a server or a client, as specified. The shared structure of the device associated with cluster must also be specified.

The function must be the first OTA function called in the application, and must be called after the stack has been started and after the ZCL has been initialized.

Parameters

- *psClusterInstance*: Pointer to structure containing information about the cluster instance to be created (see Section 6.1.16)
- *bIsServer*: Side of cluster to be implemented on this device:
  - TRUE - Server
  - FALSE - Client
- *psClusterDefinition*: Pointer to structure indicating the type of cluster (see Section 6.1.2) - this structure must contain the details of the OTA Upgrade cluster
- *pvEndPointSharedStructPtr*: Pointer to shared device structure for relevant endpoint (depends on device type, e.g. Door Lock)
- *u8Endpoint*: Number of endpoint with which cluster will be associated
- *pu8AttributeControlBits*: Pointer to an array of bitmaps, one for each attribute in the relevant cluster - for internal cluster definition use only, array should be initialised to 0
- *tpsCustomDataStruct*: Pointer to structure containing custom data for OTA Upgrade cluster (see Section 49.11.2)

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL

49.10.1.2 vOTA_FlashInit

```
void vOTA_FlashInit(void *pvFlashTable,
        tsNvmDefs *psNvmStruct);
```

Description

This function initializes the Flash memory device to be used by the OTA Upgrade cluster. Information about the device must be provided, such as the device type and sector size.

If a custom or unsupported Flash memory device is used then user-defined callback functions must be provided to perform Flash memory read, write, erase and initialization operations (if an NXP-supported device is used, standard callback functions will be used):

- A general set of functions (for use by all software components) can be specified through *pvFlashTable*.
- Optionally, an additional set of functions specifically for use by the OTA Upgrade cluster can be specified in the structure referenced by *psNvmStruct*.

This function must be called after the OTA Upgrade cluster has been created (after **eOTA_Create()** has been called either directly or indirectly) and before any other OTA Upgrade functions are called.

Parameters

pvFlashTable: Pointer to general set of callback functions to perform Flash memory read, write, erase and initialization operations. If using an NXP-supported Flash memory device, set a null pointer to use standard callback functions

psNvmStruct: Pointer to structure containing information on Flash memory device - see Section 49.11.4

Returns

None

49.10.1.3 eOTA_AllocateEndpointOTASpace

```
teZCL_Status eOTA_AllocateEndpointOTASpace(
        uint8 u8Endpoint,
        uint8 *pu8Data,
        uint8 u8NumberOfImages,
        uint8 u8MaxSectorsPerImage,
        bool_t bIsServer,
        uint8 *pu8CAPublicKey);
```

Description

This function is used to allocate Flash memory space to store application images as part of the OTA upgrade process for the specified endpoint. The maximum number of images that are held at any one time must be specified as well the Flash memory start sector of every image. The maximum number of sectors used to store an image must also be specified.

The start sectors of the image space allocations are provided in an array. The index of an element of this array will subsequently be used to identify the stored image in other function calls.

Advice about the allocation of internal Flash memory space to OTA upgrade images on the client is provided in Appendix E.2

Parameters

- *u8Endpoint*: Number of endpoint for which Flash memory space is to be allocated
- *pu8Data*: Pointer to array containing the Flash memory start sector of each image (array index identifies image)
- *u8NumberOfImages*: Maximum number of application images that are stored in Flash memory at any one time
- *u8MaxSectorsPerImage*: Maximum number of sectors to be used to store an individual application image
- *bIsServer*: Side of cluster implemented on this device:
- TRUE - Server
- FALSE - Client
- *pu8CAPublicKey*: Pointer to Certificate Authority public key (provided in the security certificate from a company such as Certicom)

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL
- E_ZCL_ERR_INVALID_VALUE
- E_ZCL_ERR_EP_RANGE
- E_ZCL_ERR_CLUSTER_NOT_FOUND
- E_ZCL_ERR_PARAMETER_NULL

49.10.1.4 vOTA_GenerateHash

```
void vOTA_GenerateHash(
        tsZCL_EndPointDefinition *psEndPointDefinition,
        tsOTA_Common *psCustomData,
        bool bIsServer,
```

```
        bool bHeaderPresent,
        AESSW_Block_u *puHash,
        uint8 u8ImageLocation);
```

Description

This function can be used to generate a hash checksum for an application image in Flash memory, using the Matyas-Meyer-Oseas cryptographic hash.

Parameters

psEndPointDefinition Pointer to structure which defines endpoint corresponding to the application (see Section 6.1.1)
psCustomData Pointer to data structure connected with event associated with the checksum (see Section 49.11.2)
bIsServer Side of cluster implemented on this device:
TRUE - Server
FALSE - Client
bHeaderPresent Presence of image header:
TRUE - Present
FALSE - Absent
puHash Pointer to structure to receive calculated hash checksum
u8ImageLocation Number of sector where image starts in Flash memory

Returns

None

49.10.1.5 eOTA_GetCurrentOtaHeader

```
teZCL_Status eOTA_GetCurrentOtaHeader(
                uint8 u8Endpoint,
        bool_t bIsServer,
        tsOTA_ImageHeader *psOTAHeader);
```

Description

This function can be used to obtain the OTA header of the application image which is currently running on the local node.

The obtained parameter values are received in a `tsOTA_ImageHeader` structure.

Parameters

u8Endpoint Number of endpoint on which cluster operates
bIsServer Side of the cluster implemented on this device:
TRUE - Server
FALSE - Client
psOTAHeader Pointer to structure to receive the current OTA header (see Section 49.11.1)

Returns

E_ZCL_SUCCESS
E_ZCL_ERR_EP_RANGE
E_ZCL_ERR_CLUSTER_NOT_FOUND
E_ZCL_ERR_PARAMETER_NULL

49.10.2 Server Functions

The following OTA Upgrade cluster functions can be used on the cluster server only:

1. eOTA_SetServerAuthorisation
2. eOTA_SetServerParams
3. eOTA_GetServerData
4. eOTA_EraseFlashSectorsForNewImage
5. eOTA_FlashWriteNewImageBlock
6. eOTA_NewImageLoaded
7. eOTA_ServerImageNotify
8. eOTA_ServerQueryNextImageResponse
9. eOTA_ServerImageBlockResponse
10. eOTA_SetWaitForDataParams
11. eOTA_ServerUpgradeEndResponse
12. eOTA_ServerSwitchToNewImage
13. eOTA_InvalidateStoredImage
14. eOTA_ServerQuerySpecificFileResponse

49.10.2.1 eOTA_SetServerAuthorisation

```
teZCL_Status eOTA_SetServerAuthorisation(
        uint8 u8Endpoint,
        eOTA_AuthorisationState eState,
        uint64 *pu64WhiteList,
        uint8 u8Size);
```

Description

This function can be used to define a set of clients to which the server is authorized to download application images. The function allows all clients to be authorized or a list of selected authorized clients to be provided. Clients are specified in this list by means of their 64-bit IEEE/MAC addresses.

Parameters

- *u8Endpoint*: Number of endpoint (on server) on which cluster operates
- *eState*: Indicates whether a list of authorized clients is used or all clients are authorized - one of:
  - E_CLD_OTA_STATE_USE_LIST
  - E_CLD_OTA_STATE_ALLOW_ALL
- *pu64WhiteList*: Pointer to list of IEEE/MAC addresses of authorized clients (ignored if all clients are authorized through *eState* parameter)
- *u8Size* Number of clients in list
- (ignored if all clients are authorized through *eState* parameter)

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL

49.10.2.2 eOTA_SetServerParams

```
teZCL_Status eOTA_SetServerParams(
: uint8 u8Endpoint,
: uint8 u8ImageIndex,
: tsCLD_PR_Ota *psOTAData);
```

Description

This function can be used to set server parameter values (including query jitter, data size, image data, current time and upgrade time) for a particular image stored on the server. The parameter values to be set are specified in a structure, described in Section 49.11.22. For detailed descriptions of these parameters, refer to the *ZigBee Over-the-Air Upgrading Cluster Specification (095264)* from the ZigBee Alliance.

If this function is not called, default values are used for these parameters.

The current values of these parameters can be obtained using the function **eOTA_GetServerData()**.

The index of the image for which server parameter values are to be set must be specified. For an image stored in Flash memory, this index will take a value in the range 0 to (OTA_MAX_IMAGES_PER_ENDPOINT - 1).

Parameters

u8Endpoint: Number of endpoint (on server) on which cluster operates
u8ImageIndex: Index number of image
psOTAData: Pointer to structure containing parameter values to be set (see Section 49.11.22)

Returns

E_ZCL_SUCCESS
E_ZCL_FAIL

49.10.2.3 eOTA_GetServerData

```
teZCL_Status eOTA_GetServerData(
        uint8 u8Endpoint,
        uint8 u8ImageIndex,
        tsCLD_PR_Ota *psOTAData);
```

Description

This function can be used to obtain server parameter values (including query jitter, data size, image data, current time and upgrade time). The obtained parameter values are received in a structure, described in Section 49.11.22. For detailed descriptions of these parameters, refer to the *ZigBee Over-the-Air Upgrading Cluster Specification (095264)* from the ZigBee Alliance.

The values of these parameters can be set by the application using the function **eOTA_SetServerParams()**.

The index of the image for which server parameter values are to be obtained must be specified. For an image stored in the Flash memory, this index will take a value in the range 0 to (OTA_MAX_IMAGES_PER_ENDPOINT - 1).

Parameters

u8Endpoint: Number of endpoint (on server) on which cluster operates
u8ImageIndex: Index number of image
psOTAData: Pointer to structure to receive parameter values (see Section 49.11.22)

Returns

E_ZCL_SUCCESS
E_ZCL_FAIL

49.10.2.4 eOTA_EraseFlashSectorsForNewImage

```
teZCL_Status eOTA_EraseFlashSectorsForNewImage(
        uint8 u8Endpoint,
        uint8 u8ImageIndex);
```

Description

This function can be used to erase certain sectors of the Flash memory of the device in the OTA server node. The sectors allocated to the specified image index number will be erased so that the sectors (and index number) can be re-used. The function is normally called before writing a new upgrade image to Flash memory.

The specified image index number must be in the range 0 to (OTA_MAX_IMAGES_PER_ENDPOINT - 1).

Parameters

u8Endpoint: Number of endpoint (on server) on which cluster operates
u8ImageIndex: Index number of image

Returns

E_ZCL_ERR_EP_RANGE
E_ZCL_ERR_PARAMETER_NULL
E_ZCL_ERR_CLUSTER_NOT_FOUND
E_ZCL_SUCCESS

49.10.2.5 eOTA_FlashWriteNewImageBlock

```
teZCL_Status eOTA_FlashWriteNewImageBlock(
        uint8 u8Endpoint,
        uint8 u8ImageIndex,
        bool bIsServerImage,
        uint8 *pu8UpgradeBlockData,
        uint8 u8UpgradeBlockDataLength,
        uint32 u32FileOffSet);
```

Description

This function can be used to write a block of an upgrade image to the devices internal Flash memory in the OTA server node. The image may be either of the following:

- An upgrade image for the server itself (the server will later be rebooted from this image)
- An upgrade image for one or more clients, which will later be made available for OTA distribution through the wireless network

The image in Flash memory to which the block belongs is identified by its index number. The specified image index number must be in the range 0 to (OTA_MAX_IMAGES_PER_ENDPOINT - 1).

Note that for JN518x, K32W041, and K32W061, internal Flash memory, writes must be 512-byte aligned.

Parameters

u8Endpoint: Number of endpoint (on server) on which cluster operates
u8ImageIndex: Index number of image
bIsServerImage: Indicates whether new image is for the server or a client:
TRUE - Server image
FALSE - Client image
pu8UpgradeBlockData: Pointer to image block to be written
u8UpgradeBlockDataLength: Size, in bytes, of image block to be written
u32FileOffSet: Offset of block from start of image file (in terms of number of bytes)

Returns

E_ZCL_ERR_EP_RANGE
E_ZCL_ERR_PARAMETER_NULL
E_ZCL_ERR_CLUSTER_NOT_FOUND
E_ZCL_FAIL
E_ZCL_SUCCESS

49.10.2.6 eOTA_NewImageLoaded

```
teZCL_Status eOTA_NewImageLoaded(
        uint8 u8Endpoint,
        bool bIsImageOnCoProcessorMedia,
        tsOTA_CoProcessorOTAHeader
 *psOTA_CoProcessorOTAHeader);
```

Description

This function can be used for two purposes which relate to a new application image and which depend on whether the image has been stored in the internal Flash memory of the device or in the external storage device of a co-processor (if any) within the server node:

- For an image stored in internal Flash memory, the function can be used to notify the OTA Upgrade cluster server on the specified endpoint that a new application image has been loaded into Flash memory and is available for download to clients. The server then validates the new image.
- For one or more images stored in the co-processor's external storage device, the function can be used to provide OTA header information for the image(s) to the cluster server. In the case of more than one image

stored in co-processor storage, this function may replicate OTA header information for older images already registered with the server.

Note: *The co-processor option is currently not supported for JN518x, K32W041, or K32W061.*

Parameters

u8Endpoint: Number of endpoint (on server) on which cluster operates
bIsImageOnCoProcessorMedia: Flag indicating whether image is stored in co-processor external storage device:
TRUE - Stored in co-processor
FALSE - Stored in internal Flash memory
psOTA_CoProcessorOTAHeader: Pointer to OTA headers of images which are held in co-processor storage device

Returns

E_ZCL_SUCCESS
E_ZCL_FAIL

49.10.2.7 eOTA_ServerImageNotify

```
teZCL_Status eOTA_ServerImageNotify(
        uint8 u8SourceEndpoint,
        uint8 u8DestinationEndpoint,
        tsZCL_Address *psDestinationAddress,
        tsOTA_ImageNotifyCommand *psImageNotifyCommand);
```

Description

This function issues an Image Notify message to one or more clients to indicate that a new application image is available for download.

The message can be unicast to an individual client or multicast to selected clients (but cannot be broadcast to all clients, for security reasons).

Parameters

u8SourceEndpoint: Number of endpoint (on server) from which the message is sent
u8DestinationEndpoint: Number of endpoint (on client) to which the message is sent
psDestinationAddress: Pointer to structure containing the address of the target client for the message - a multicast to more than one client is also possible (see Section 6.1.4)
psImageNotifyCommand: Pointer to structure containing payload for message (see Section 49.11.5)

Returns

E_ZCL_SUCCESS
E_ZCL_FAIL

49.10.2.8 eOTA_ServerQueryNextImageResponse

```
teZCL_Status eOTA_ServerQueryNextImageResponse(
```

```
        uint8 u8SourceEndpoint,
        uint8 u8DestinationEndpoint,
        tsZCL_Address *psDestinationAddress,
        tsOTA_QueryImageResponse
 *psQueryImageResponsePayload,
        uint8 u8TransactionSequenceNumber);
```

Description

This function issues a Query Next Image Response to a client which has sent a Query Next Image Request (the arrival of this request triggers the event E_CLD_OTA_COMMAND_QUERY_NEXT_IMAGE_REQUEST on the server).

The Query Next Image Response contains information on the latest application image available for download to the client, including the image size and file version.

Note: *The cluster server responds automatically to a Query Next Image Request, so it is not normally necessary for the application to call this function.*

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 is 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

u8SourceEndpoint: Number of endpoint (on server) from which the response is sent
u8DestinationEndpoint: Number of endpoint (on client) to which the response is sent
psDestinationAddress: Pointer to structure containing the address of the target client for the response (see Section 6.1.4)
psQueryImageResponsePayload: Pointer to structure containing payload for response (see Section 49.11.7)
u8TransactionSequenceNumber: Pointer to a location to store the Transaction Sequence Number (TSN) of the request

Returns

E_ZCL_SUCCESS
E_ZCL_FAIL

49.10.2.9 eOTA_ServerImageBlockResponse

```
teZCL_Status eOTA_ServerImageBlockResponse(
        uint8 u8SourceEndpoint,
        uint8 u8DestinationEndpoint,
        tsZCL_Address *psDestinationAddress,
        tsOTA_ImageBlockResponsePayload
  *psImageBlockResponsePayload,
        uint8 u8BlockSize,
        uint8 u8TransactionSequenceNumber);
```

Description

This function issues an Image Block Response, containing a block of image data, to a client to which the server is downloading an application image. The function is called after receiving an Image Block Request from the

client, indicating that the client is ready to receive the next block of the application image (the arrival of this request triggers the event E_CLD_OTA_COMMAND_BLOCK_REQUEST on the server).

The size of the block, in bytes, is specified as part of the function call. This must be less than or equal to the maximum possible block size defined in the **zcl_options.h** file (see Section 49.13).

Note: *The cluster server responds automatically to an Image Block Request, so it is not normally necessary for the application to call this function.*

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 is 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

u8SourceEndpoint: Number of endpoint (on server) from which the response is sent
u8DestinationEndpoint: Number of endpoint (on client) to which the response is sent
psDestinationAddress: Pointer to structure containing the address of the target client for the response (see Section 6.1.4)
psImageBlockResponsePayload: Pointer to structure containing payload for response (see Section 49.11.10)
u8BlockSize: Size, in bytes, of block to be transferred
u8TransactionSequenceNumber: Pointer to a location to store the Transaction Sequence Number (TSN) of the request

Returns

E_ZCL_SUCCESS
E_ZCL_FAIL

49.10.2.10 eOTA_SetWaitForDataParams

```
teZCL_Status eOTA_SetWaitForDataParams(
        uint8 u8Endpoint,
        uint16 u16ClientAddress,
        tsOTA_WaitForData *sWaitForDataParams);
```

Description

This function can be used to send an Image Block Response with a status of OTA_STATUS_WAIT_FOR_DATA to a client, in response to an Image Block Request from the client.

The payload of this response includes a new value for the ‘block request delay’ attribute on the client. This value can be used by the client for ‘rate limiting’ -that is, to control the rate at which the client requests data blocks from the server and therefore the average OTA download rate from the server to the client.

Rate limiting is described in more detail in Section 49.8.1.

Parameters

u8Endpoint: Number of endpoint (on server) from which the response is sent
u16ClientAddress: Network address of client device to which the response is sent
sWaitForDataParams: Pointer to structure containing ‘Wait for Data’ parameter values for Image Block Response payload (see Section 49.11.15)

Returns

E_ZCL_SUCCESS
E_ZCL_FAIL

49.10.2.11 eOTA_ServerUpgradeEndResponse

```
teZCL_Status eOTA_ServerUpgradeEndResponse(
        uint8 u8SourceEndpoint,
        uint8 u8DestinationEndpoint,
        tsZCL_Address *psDestinationAddress,
        tsOTA_UpgradeEndResponsePayload
 *psUpgradeResponsePayload,
        uint8 u8TransactionSequenceNumber);
```

Description

This function issues an Upgrade End Response to a client to which the server has been downloading an application image. The function is called after receiving an Upgrade End Request from the client, indicating that the client has received the entire application image and verified it (the arrival of this request triggers the event E_CLD_OTA_COMMAND_UPGRADE_END_REQUEST on the server).

The Upgrade End Response includes the upgrade time for the downloaded image as well as the current time (the client will use this information to implement a delay before upgrading the running application image).

Note: *The cluster server responds automatically to an Upgrade End Request, so it is not normally necessary for the application to call this function.*

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 is 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

u8SourceEndpoint: Number of endpoint (on server) from which the response is sent
u8DestinationEndpoint: Number of endpoint (on client) to which the response is sent
psDestinationAddress: Pointer to structure containing the address of the target client for the response (see Section 6.1.4)
psUpgradeResponsePayload: Pointer to structure containing payload for response (see Section 49.11.12)
u8TransactionSequenceNumber: Pointer to a location to store the Transaction Sequence Number (TSN) of the request

Returns

E_ZCL_SUCCESS
E_ZCL_FAIL

49.10.2.12 eOTA_ServerSwitchToNewImage

```
teZCL_Status eOTA_ServerSwitchToNewImage(
        uint8 u8Endpoint,
        uint8 u8ImageIndex);
```

Description

This function can be used to force a reset of the device in the OTA server node and, on reboot, run a new application image that has been saved in the attached Flash memory.

Before forcing the reset of the remove device, the function checks whether the version of the new image is greater than the version of the current image. If this is the case, the function invalidates the currently running image in Flash memory and initiates a software reset - otherwise, it returns an error.

The new application image is identified by its index number. The specified image index number must be in the range 0 to (OTA_MAX_IMAGES_PER_ENDPOINT - 1).

Parameters

u8Endpoint: Number of endpoint (on server) on which cluster operates
u8ImageIndex: Index number of image

Returns

E_ZCL_ERR_EP_RANGE
E_ZCL_ERR_CLUSTER_NOT_FOUND
E_ZCL_FAIL
E_ZCL_SUCCESS

49.10.2.13 eOTA_InvalidateStoredImage

```
teZCL_Status eOTA_InvalidateStoredImage(
        uint8 u8Endpoint,
        uint8 u8ImageIndex);
```

Description

This function can be used to invalidate an application image that is held in the Flash memory of the device. Once the image has been invalidated, it will no longer to available for OTA upgrade.

The image to be invalidated is identified by its index number. The specified image index number must be in the range 0 to (OTA_MAX_IMAGES_PER_ENDPOINT - 1).

Parameters

u8Endpoint: Number of endpoint (on server) on which cluster operates
u8ImageIndex: Index number of image to be invalidated

Returns

E_ZCL_ERR_EP_RANGE
E_ZCL_ERR_PARAMETER_NULL
E_ZCL_ERR_CLUSTER_NOT_FOUND
E_ZCL_SUCCESS

49.10.2.14 eOTA_ServerQuerySpecificFileResponse

```
teZCL_Status eOTA_ServerQuerySpecificFileResponse(
```

```
        uint8 u8SourceEndpoint,
        uint8 u8DestinationEndpoint,
        tsZCL_Address *psDestinationAddress,
        tsOTA_QuerySpecificFileResponsePayload
 *psQuerySpecificFileResponsePayload,
        uint8 u8TransactionSequenceNumber);
```

Description

This function can be used to issue a Query Specific File Response to a client which has sent a Query Specific File Request (the arrival of this request triggers the event E_CLD_OTA_COMMAND_QUERY_SPECIFIC_FILE_REQUEST on the server). The Query Specific File Response contains information on the latest device-specific file available for download to the client, including the file size and file version.

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 is 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

u8SourceEndpoint: Number of endpoint (on server) from which the response is sent
u8DestinationEndpoint: Number of endpoint (on client) to which the response is sent
psDestinationAddress: Pointer to structure containing the address of the target client
psQuerySpecificFileResponsePayload:
Pointer to structure containing payload for Query Specific File Response (see Section 49.11.20)
u8TransactionSequenceNumber: Pointer to a location to store the Transaction Sequence Number (TSN) of the request

Returns

E_ZCL_SUCCESS
E_ZCL_FAIL

49.10.3 Client Functions

The following OTA Upgrade cluster functions can be used on the cluster client only:

1. eOTA_SetServerAddress
2. eOTA_ClientQueryNextImageRequest
3. eOTA_ClientImageBlockRequest
4. eOTA_ClientImagePageRequest
5. eOTA_ClientUpgradeEndRequest
6. eOTA_HandleImageVerification
7. eOTA_UpdateClientAttributes
8. eOTA_RestoreClientData
9. vOTA_SetImageValidityFlag
10. eOTA_ClientQuerySpecificFileRequest
11. eOTA_SpecificFileUpgradeEndRequest
12. vOTA_SetLowVoltageFlag

49.10.3.1 eOTA_SetServerAddress

```
teZCL_Status eOTA_SetServerAddress(
        uint8 u8Endpoint,
        uint64 u64IeeeAddress,
        uint16 u16ShortAddress);
```

Description

This function sets the addresses (64-bit IEEE/MAC address and 16-bit network address) of the OTA Upgrade cluster server that will be used to provide application upgrade images to the local client.

The function should be called after a server discovery has been performed to find a suitable server - this is done by sending out a Match Descriptor Request using the function **ZPS_eAplZdpMatchDescRequest()** described in the *ZigBee 3.0 Stack User Guide (JNUG3130)*. The server discovery must be completed and a server address set before any OTA-related message exchanges can occur (e.g. image request).

Parameters

- *u8Endpoint*: Number of endpoint corresponding to application
- *u64IeeeAddress*: IEEE/MAC address of server
- *u16ShortAddress*: Network address of server

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL

49.10.3.2 eOTA_ClientQueryNextImageRequest

```
teZCL_Status eOTA_ClientQueryNextImageRequest(
        uint8 u8SourceEndpoint,
        uint8 u8DestinationEndpoint,
        tsZCL_Address *psDestinationAddress,
        tsOTA_QueryImageRequest
  *psQueryImageRequest);
```

Description

This function issues a Query Next Image Request to the server and should be called in either of the following situations:

- to poll for a new application image (typically used in this way by an End Device) - in this case, the function should normally be called periodically
- to respond to an Image Notify message from the server, which indicated that a new application image is available for download - in this case, the function call should be prompted by the event E_CLD_OTA_COMMAND_IMAGE_NOTIFY

The payload of the request includes the relevant image type, current file version, hardware version and manufacturer code.

As a result of this function call, a Query Next Image Response will (eventually) be received from the server. The arrival of this response will trigger an E_CLD_OTA_COMMAND_QUERY_NEXT_IMAGE_RESPONSE event.

Parameters

- *u8SourceEndpoint*: Number of endpoint (on client) from which the request is sent
- *u8DestinationEndpoint*: Number of endpoint (on server) to which the request is sent
- *psDestinationAddress*: Pointer to structure containing the address of the target server (see Section 6.1.4)
- *psQueryImageRequest*: Pointer to structure containing payload for request (see Section 49.11.6)

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL

49.10.3.3 eOTA_ClientImageBlockRequest

```
teZCL_Status eOTA_ClientImageBlockRequest(
        uint8 u8SourceEndpoint,
        uint8 u8DestinationEndpoint,
        tsZCL_Address *psDestinationAddress,
        tsOTA_BlockRequest
 *psOtaBlockRequest);
```

Description

This function can be used during an image download to send an Image Block Request to the server, in order to request the next block of image data.

As a result of this function call, an Image Block Response containing the requested data block will (eventually) be received from the server. The arrival of this response will trigger an E_CLD_OTA_COMMAND_QUERY_NEXT_IMAGE_RESPONSE event.

Note: *The cluster client automatically sends Image Block Requests to the server during a download, so it is not normally necessary for the application to call this function.*

Parameters

- *u8SourceEndpoint*: Number of endpoint (on client) from which the request is sent
- *u8DestinationEndpoint*: Number of endpoint (on server) to which the request is sent
- *psDestinationAddress*: Pointer to structure containing the address of the target server (see Section 6.1.4)
- *psOtaBlockRequest*: Pointer to structure containing payload for request (see Section 49.11.8)

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL

49.10.3.4 eOTA_ClientImagePageRequest

```
teZCL_Status eOTA_ClientImagePageRequest(
        uint8 u8SourceEndpoint,
        uint8 u8DestinationEndpoint,
        tsZCL_Address *psDestinationAddress,
        tsOTA_ImagePageRequest *psOtaPageRequest);
```

Description

This function can be used during an image download to send an Image Page Request to the server, in order to request the next page of image data. In this function call, a structure must be supplied which contains the payload data for the request. This data includes the page size, in bytes.

***Note:** **Note 1:** Image Page Requests can be used instead of Image Block Requests if page requests have been enabled in the **zcl_options.h** file for the client and server (see Section 49.13).*

***Note:** **Note 2:** The cluster client automatically sends Image Page Requests (if enabled) to the server during a download, so it is not normally necessary for the application to call this function.*

As a result of this function call, a sequence of Image Block Responses containing the requested data will (eventually) be received from the server. The arrival of each response will trigger an E_CLD_OTA_COMMAND_BLOCK_RESPONSE event on the client. If this function is used (rather than the stack) to issue Image Page Requests, it is the responsibility of the application to keep a count of the number of data bytes received since the Image Page Request was issued - when all the requested page data has been received, this count will equal the specified page size.

Page requests are described in more detail Section 49.8.4.

Parameters

- *u8SourceEndpoint*: Number of endpoint (on client) from which the request is sent
- *u8DestinationEndpoint*: Number of endpoint (on server) to which the request is sent
- *psDestinationAddress*: Pointer to structure containing the address of the target server (see Section 6.1.4)
- *psOtaPageRequest*: Pointer to structure containing payload for request (see Section 49.11.9)

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL

49.10.3.5 eOTA_ClientUpgradeEndRequest

```
teZCL_Status eOTA_ClientUpgradeEndRequest(
        uint8 u8SourceEndpoint,
        uint8 u8DestinationEndpoint,
        tsZCL_Address *psDestinationAddress,
        tsOTA_UpgradeEndRequestPayload
 *psUpgradeEndRequestPayload);
```

Description

This function can be used during an image download to send an Upgrade End Request to the server. This is normally used to indicate that all the image data has been received and that the image has been successfully verified - it is the responsibility of the client to determine when all the image data has been received (using the image size quoted in the original Query Next Image Response) and then to verify the image.

In addition to the status OTA_STATUS_SUCCESS described above, the function can be used by the client to report other conditions to the server:

- OTA_REQUIRE_MORE_IMAGE: The downloaded image was successfully received and verified, but the client requires multiple images before performing an upgrade
- OTA_STATUS_INVALID_IMAGE: The downloaded image failed the verification checks and will be discarded

- OTA_STATUS_ABORT The image download that is currently in progress should be cancelled

In all three of the above cases, the client may then request another download.

When the function is called to report success, an Upgrade End Response will (eventually) be received from the server, indicating when the image upgrade should be implemented (a time delay may be indicated in the response). The arrival of this response will trigger an E_CLD_OTA_COMMAND_UPGRADE_END_RESPONSE event.

Note: *The cluster client automatically sends an Upgrade End Request to the server on completion of a download, so it is not normally necessary for the application to call this function.*

Parameters

- *u8SourceEndpoint*: Number of endpoint (on client) from which the request is sent
- *u8DestinationEndpoint*: Number of endpoint (on server) to which the request is sent
- *psDestinationAddress*: Pointer to structure containing the address of the target server (see Section 6.1.4)
- *psUpgradeEndRequestPayload*: Pointer to structure containing payload for request, including reported status (see Section 49.11.11)

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL

49.10.3.6 eOTA_HandleImageVerification

```
teZCL_Status eOTA_HandleImageVerification(
uint8 u8SourceEndPointId,
uint8 u8DstEndpoint,
teZCL_Status eImageVerificationStatus);
```

Description

This function transmits an upgrade end request with the specified status.

Parameters

- *u8SourceEndPointId*: Identifier of endpoint on which the cluster client operates
- *u8DstEndpoint*: Identifier of endpoint (on the server) to which the upgrade end request is sent
- *eImageVerificationStatus*: Image status code

Returns

- E_ZCL_FAIL
- E_ZCL_SUCCESS

49.10.3.7 eOTA_UpdateClientAttributes

```
teZCL_Status eOTA_UpdateClientAttributes(
        uint8 u8Endpoint);
```

Description

This function can be used on a client to set the OTA Upgrade cluster attributes to their default values. It should be called during application initialization after the cluster instance has been created using **eOTA_Create()**.

Following subsequent resets, provided that context data has been saved, the application should call **eOTA_RestoreClientData()** instead of this function.

Parameters

- *u8Endpoint*: Number of endpoint corresponding to context data

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL

49.10.3.8 eOTA_RestoreClientData

```
teZCL_Status eOTA_RestoreClientData(
        uint8 u8Endpoint,
        tsOTA_PersistedData *psOTAData,
        bool_t bReset);
```

Description

This function can be used to restore OTA Upgrade context data that has been previously saved to Flash memory (using the NVM) on the local client - for example, it restores the OTA Upgrade attribute values. The function can be used to restore the data in RAM following a device reset or simply to refresh the data in RAM.

Parameters

- *8Endpoint*: Number of endpoint corresponding to context data
- *psOTAData*: Pointer to structure containing the context data to be restored (see Section 49.11.13)
- *bReset*: Indicates whether the data restoration follows a reset:
- TRUE - Follows a reset
- FALSE - Does not follow a reset

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL

49.10.3.9 vOTA_SetImageValidityFlag

```
void vOTA_SetImageValidityFlag(
        uint8 u8Location,
                    tsOTA_Common *psCustomData,
                    bool bSet,
                    tsZCL_EndPointDefinition *psEndPointDefinition);
```

Description

This function can be used to set an image validity flag once a downloaded upgrade image has been received and verified by the client.

Parameters

- *u8Location*: Number of sector where image starts in Flash memory
- *psCustomData*: Pointer to custom data for image (see Section 49.11.2)
- *bSet*: Flag state to be set:
- TRUE - Reset
- FALSE - No reset
- *psEndPointDefinition*: Pointer to endpoint definition (see Section 6.1.1)

Returns

- None

49.10.3.10 eOTA_ClientQuerySpecificFileRequest

```
eOTA_ClientQuerySpecificFileRequest(
        uint8 u8SourceEndpoint,
        uint8 u8DestinationEndpoint,
        tsZCL_Address *psDestinationAddress,
        tsOTA_QuerySpecificFileRequestPayload
 *psQuerySpecificFileRequestPayload);
```

Description

This function can be used to issue a Query Specific File Request to the server. It should be called to request a device-specific file from the server. As a result of this function call, a Query Specific File Response will (eventually) be received in reply.

Parameters

- *u8SourceEndpoint*: Number of endpoint (on client) from which the request is sent
- *u8DestinationEndpoint*: Number of endpoint (on server) to which the request is sent
- *psDestinationAddress*: Pointer to structure containing the address of the target server
- *psQuerySpecificFileRequestPayload*: Pointer to structure containing payload for Query Specific File Request

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL

49.10.3.11 eOTA_SpecificFileUpgradeEndRequest

```
eOTA_SpecificFileUpgradeEndRequest(
                uint8 u8SourceEndPointId,
                uint8 u8Status);
```

Description

This function can be used to issue an Upgrade End Request for the device-specific file download that is in progress in order to indicate to the server that the download has completed. This request can be issued by the client optionally after the downloaded image has been verified and found to be valid.

Parameters

- *u8SourceEndPointId*: Number of endpoint (on client) from which the request is sent
- *u8Status*: Download status of device-specific file - if the file has been completely and successfully received, this parameter must be set to OTA_STATUS_SUCCESS

Returns

- E_ZCL_SUCCESS
- E_ZCL_FAIL

49.10.3.12 vOTA_SetLowVoltageFlag

```
void vOTA_SetLowVoltageFlag(bool bValue);
```

Description

This function can be used to configure the low-voltage flag on a node hosting an OTA Upgrade cluster client. This flag should be set when the supply voltage to the underlying hardware is below that required for normal operation and the node should not participate in an OTA upgrade.

- When the flag is set, the client stops sending Image Block Requests to the server
- When the flag is cleared, the client resumes sending Image Block Requests to the server

Use of the low-voltage flag must be enabled at compile-time by including the macro OTA_UPGRADE_VOLTAGE_CHECK in the **zcl_options.h** file.

Use of the low-voltage flag is described further in Section 49.8.7.

Parameters

- *bValue*: Determines the state of the low-voltage flag, as follows:
  - TRUE - Sets the flag
  - FALSE - Clears the flag

Returns

- None

49.11 Structures

49.11.1 tsOTA_ImageHeader

The following structure contains information for the OTA header:

```
typedef struct
{
```

```
            uint32 u32FileIdentifier;
    uint16 u16HeaderVersion;
    uint16 u16HeaderLength;
    uint16 u16HeaderControlField;
    uint16 u16ManufacturerCode;
    uint16 u16ImageType;
    uint32 u32FileVersion;
    uint16 u16StackVersion;
    uint8  stHeaderString[OTA_HEADER_STRING_SIZE];
            uint32 u32TotalImage;
    uint8  u8SecurityCredVersion;
    uint64 u64UpgradeFileDest;
    uint16 u16MinimumHwVersion;
    uint16 u16MaxHwVersion;
}tsOTA_ImageHeader;
```

where:

- `u32FileIdentifier` is a 4-byte value equal to 0x0BEEF11E which indicates that the file contains an OTA upgrade image
- `u16HeaderVersion` is the version of the OTA header expressed as a 2-byte value in which the most significant byte contains the major version number and the least significant byte contains the minor version number
- `u16HeaderLength` is the full length of the OTA header, in bytes
- `u16HeaderControlField` is a bitmap indicating certain information about the file, as detailed in table below.

Table 118. u16HeaderControlField bitmap

Bit	Information
0	Security credential version (in OTA header): 1: Field present in header 0: Field not present in header
1	Device-specific file (also see `u64UpgradeFileDest`): 1: File is device-specific 0: File is not device-specific
2	Maximum and minimum hardware version (in OTA header): 1: Field present in header 0: Field not present in header
3-15	Reserved

- `u16ManufacturerCode` is the ZigBee-assigned manufacturer code (0xFFFF is a wildcard value, representing any manufacturer)
- `u16ImageType` is a unique value representing the image type, where this value is normally manufacturer-specific but certain values have been reserved for specific file types, as indicated below (the wildcard value of 0xFFFF represents any file type):

Table 119. u16ImageType values

Value	File Type
0x0000 – 0xFFBF	Manufacturer-specific
0xFFC0	Security credential
0xFFC1	Configuration

Table 119. u16ImageType values*...continued*

Value	File Type
0xFFC2	Log
0xFFC3 – 0xFFFE	Reserved
0xFFFF	Wildcard

- `u32FileVersion` contains the release and build numbers of the application and stack used to produce the application image - for details of the file version format, refer to the *ZigBee Over-the-Air Upgrading Cluster Specification (095264)*
- `u16StackVersion` contains ZigBee stack version that is used by the application (this is 0x0002 for ZigBee PRO)
- `stHeaderString[]` is a manufacturer-specific string that can be used to store any useful human-readable information
- `u32TotalImage` is the total size, in bytes, of the image that will be transferred over-the air (including the OTA header and any optional data)
- `u8SecurityCredVersion` indicates the security credential version type that is required by the client in order to install the image - the possibilities are SE1.0 (0x0), SE1.1 (0x1) and SE2.0 (0x2)
- `u64UpgradeFileDest` contains the IEEE/MAC address of the destination device for the file, in the case when the file is device-specific (as indicated by bit 1 of `u16HeaderControlField`)
- `u16MinimumHwVersion` indicates the earliest hardware platform on which the image should be used, expressed as a 2-byte value in which the most significant byte contains the hardware version number and the least significant byte contains the revision number
- `u16MaxHwVersion` indicates the latest hardware platform on which the image should be used, expressed as a 2-byte value in which the most significant byte contains the hardware version number and the least significant byte contains the revision number

49.11.2 tsOTA_Common

The following structure contains data relating to an OTA message received by the cluster (server or client) - this data is used for callback functions and the local OTA state machine:

```
typedef struct
{
    tsZCL_ReceiveEventAddress  sReceiveEventAddress;
    tsZCL_CallBackEvent        sOTACustomCallBackEvent;
    tsOTA_CallBackMessage      sOTACallBackMessage;
} tsOTA_Common;
```

The fields are for internal use and no knowledge of them is required. The `tsOTA_CallBackMessage` structure is described in Section 49.11.21.

49.11.3 tsOTA_HwFncTable

The following structure contains pointers to callback functions to be used by the OTA Upgrade cluster to perform initialization, erase, write and read operations on Flash memory (if these functions are not specified, standard NXP functions will be used):

```
typedef struct
{
     void (*prInitHwCb)(uint8, void*);
     void (*prEraseCb) (uint8 u8Sector);
     void (*prWriteCb) (uint32 u32FlashByteLocation,
```

```
                          uint16 u16Len,
                          uint8 *pu8Data);
     void (*prReadCb)   (uint32 u32FlashByteLocation,
                          uint16 u16Len,
                          uint8 *pu8Data);
} tsOTA_HwFncTable;
```

where:

- `prInitHwCb` is a pointer to a callback function that is called after a cold or warm start to perform any initialization required for the Flash memory device
- `prEraseCb` is a pointer to a callback function that is called to erase a specified sector of Flash memory
- `prWriteCb` is a pointer to a callback function that is called to write a block of data to a sector, starting the write at a specified byte location in the sector (address zero is the start of the sector)
- `prReadCb` is a pointer to a callback function that is called to read a block of data from a sector, starting the read at a specified byte location in the sector (address zero is the start of the sector)

49.11.4 tsNvmDefs

The following structure contains information used to configure access to Flash memory:

```
typedef struct
{
    tsOTA_HwFncTable sOtaFnTable;
    uint32           u32SectorSize;
    uint8            u8FlashDeviceType;
}tsNvmDefs;
```

where:

- `sOtaFnTable` is a structure specifying the callback functions to be used by the cluster to perform initialization, erase, write and read operations on the Flash memory device (see Section 49.11.3) - if user-defined callback functions are not specified, standard NXP functions will be used
- `u32SectorSize` is the size of a sector of the Flash memory device, in bytes
- `u8FlashDeviceType` is a value indicating the type of Flash memory device, one of:
  - E_FL_CHIP_INTERNAL (Device internal Flash- default)

49.11.5 tsOTA_ImageNotifyCommand

The following structure contains the payload data for an Image Notify message issued by the server when a new upgrade image is available for download:

```
typedef struct
{
            teOTA_ImageNotifyPayloadType ePayloadType;
            uint32                       u32NewFileVersion;
            uint16                       u16ImageType;
            uint16                       u16ManufacturerCode;
            uint8                        u8QueryJitter;
}tsOTA_ImageNotifyCommand;
```

where:

- `ePayloadType` is a value indicating the type of payload of the command (enumerations are available - see Section 49.12.4)

- `u32NewFileVersion` is the file version of the client upgrade image that is currently available for download (the wild card of 0xFFFFFFFF is used to indicate that all clients should upgrade to this image)
- `u16ImageType` is a number indicating the type of image that is available for download (the wild card of 0xFFFF is used to indicate that all image types are involved)
- `u16ManufacturerCode` is a ZigBee-assigned number identifying the manufacturer to which the available image is connected (if all manufacturers are involved, this value should not be set)
- `u8QueryJitter` is a value between 1 and 100 (inclusive) which is used by the receiving client to decide whether to reply to this Image Notify message - for information on 'Query Jitter', refer to Section 49.7

49.11.6 tsOTA_QueryImageRequest

The following structure contains payload data for a Query Next Image Request issued by a client to poll the server for an upgrade image or to respond to an Image Notify message from the server:

```
typedef struct
{
    uint32 u32CurrentFileVersion;
    uint16 u16HardwareVersion;
    uint16 u16ImageType;
    uint16 u16ManufacturerCode;
    uint8  u8FieldControl;
}tsOTA_QueryImageRequest;
```

where:

- `u32CurrentFileVersion` is the file version of the application image that is currently running on the client that sent the request
- `u16HardwareVersion` is the hardware version of the client device (this information is optional - see `u8FieldControl` below)
- `u16ImageType` is a value in the range 0x0000-0xFFBF which identifies the type of image currently running on the client
- `u16ManufacturerCode` is the ZigBee-assigned number identifying the manufacturer of the client device
- `u8FieldControl` is a bitmap indicating whether certain optional information about the client is included in this Query Next Image Request message. Currently, this optional information consists only of the hardware version (contained in `u16HardwareVersion` above) - bit 0 is set to '1' if the hardware version is included or to '0' otherwise (all other bits are reserved)

49.11.7 tsOTA_QueryImageResponse

The following structure contains payload data for a Query Next Image Response issued by the server (as the result of a Query Next Image Request from a client):

```
typedef struct
{
    uint32 u32ImageSize;
    uint32 u32FileVersion;
    uint16 u16ManufacturerCode;
    uint16 u16ImageType;
    uint8  u8Status;
}tsOTA_QueryImageResponse;
```

where:

- `u32ImageSize` is the total size of the available image, in bytes
- `u32FileVersion` is the file version of the available image

- u16ManufacturerCode is the manufacturer code that was received from the client in the Query Next Image Request message
- u16ImageType is the image type that was received from the client in the Query Next Image Request message
- u8Status indicates whether a suitable image is available for download:
  - OTA_STATUS_SUCCESS: A suitable image is available
  - OTA_STATUS_NO_IMAGE_AVAILABLE: No suitable image is available

    The other elements of the structure are only included in the case of success.

49.11.8 tsOTA_BlockRequest

The following structure contains payload data for an Image Block Request issued by a client to request an image data block from the server:

```
typedef struct
{
    uint64 u64RequestNodeAddress;
    uint32 u32FileOffset;
    uint32 u32FileVersion;
    uint16 u16ImageType;
    uint16 u16ManufactureCode;
    uint16 u16BlockRequestDelay;
    uint8 u8MaxDataSize;
    uint8 u8FieldControl;
}tsOTA_BlockRequest;
```

where:

- u64RequestNodeAddress is the IEEE/MAC address of the client device from which the request originates (this information is optional - see u8FieldControl below)
- u32FileOffset specifies the offset from the beginning of the upgrade image, in bytes, of the requested data block (this value is therefore determined by the amount of image data previously received)
- u32FileVersion is the file version of the upgrade image for which a data block is being requested
- u16ImageType is a value in the range 0x0000-0xFFBF which identifies the type of image for which a data block is being requested
- u16ManufactureCode is the ZigBee-assigned number identifying the manufacturer of the client device from which the request originates
- u16BlockRequestDelay is used in ‘rate limiting’ to specify the value of the ‘block request delay’ attribute for the client - this is minimum time, in milliseconds, that the client must wait between consecutive block requests (the client will update the local attribute with this value). If the server does not support rate limiting or does not need to limit the download rate to the client, this field will be set to 0
- u8MaxDataSize specifies the maximum size, in bytes, of the data block that the client can receive in one transfer (the server must therefore not send a data block that is larger than indicated by this value)
- u8FieldControl is a bitmap indicating whether certain optional information about the client is included in this Image Block Request message. Currently, this optional information consists only of the IEEE/MAC address of the client (contained in 64RequestNodeAddress above) - bit 0 is set to ‘1’ if this address is included or to ‘0’ otherwise (all other bits are reserved)

49.11.9 tsOTA_ImagePageRequest

The following structure contains payload data for an Image Page Request issued by a client to request a page of image data (multiple blocks) from the server:

```
typedef struct
{
    uint64 u64RequestNodeAddress;
    uint32 u32FileOffset;
    uint32 u32FileVersion;
    uint16 u16PageSize;
    uint16 u16ResponseSpacing;
    uint16 u16ImageType;
    uint16 u16ManufactureCode;
    uint8 u8MaxDataSize;
    uint8 u8FieldControl;
}tsOTA_ImagePageRequest;
```

where:

- `u64RequestNodeAddress` is the IEEE/MAC address of the client device from which the request originates (this information is optional - see `u8FieldControl` below)
- `u32FileOffset` specifies the offset from the beginning of the upgrade image, in bytes, of the first data block of the requested page (this value is therefore determined by the amount of image data previously received)
- `u32FileVersion` is the file version of the upgrade image for which data is being requested
- `u16PageSize` is the total number of data bytes (in the page) to be returned by the server before the next Image Page Request can be issued (this must be larger than the value of `u8MaxDataSize` below)
- `u16ResponseSpacing` specifies the time-interval, in milliseconds, that the server should introduce between consecutive transmissions of Image Block Responses (which is sent in response to the Image Page Request)
- `u16ImageType` is a value in the range 0x0000-0xFFBF which identifies the type of image for which data is being requested
- `u16ManufactureCode` is the ZigBee-assigned number identifying the manufacturer of the client device from which the request originates
- `u8MaxDataSize` specifies the maximum size, in bytes, of the data block that the client can receive in one transfer (the server must therefore not send a data block in an Image Block Response that is larger than indicated by this value)
- `u8FieldControl` is a bitmap indicating whether certain optional information about the client is included in this Image Page Request message. Currently, this optional information consists only of the IEEE/MAC address of the client (contained in `64RequestNodeAddress` above) - bit 0 is set to ‘1’ if this address is included or to ‘0’ otherwise (all other bits are reserved)

49.11.10 tsOTA_ImageBlockResponsePayload

The following structure contains payload data for an Image Block Response issued by the server (as the result of an Image Block Request from a client):

```
typedef struct
{
            uint8 u8Status;
            union
            {
                        tsOTA_WaitForData                    sWaitForData;
         tsOTA_SuccessBlockResponsePayload   sBlockPayloadSuccess;
            }uMessage;
}tsOTA_ImageBlockResponsePayload;
```

where:

- u8Status indicates whether a data block is included in the response:
  - OTA_STATUS_SUCCESS: A data block is included
  - OTA_STATUS_WAIT_FOR_DATA: No data block is included - client should re-request a data block after a waiting time
- The element used from the union depends on the status reported above:
  - sWaitForData is a structure containing information used to instruct the requesting client to wait for a time before requesting the data block again or requesting the next data block (see Section 49.11.15) - this information is only provided in the case of the status OTA_STATUS_WAIT_FOR_DATA
  - sBlockPayloadSuccess is a structure containing a requested data block and associated information (see Section 49.11.13) - this data is only provided in the case of the status OTA_STATUS_SUCCESS

49.11.11 tsOTA_UpgradeEndRequestPayload

The following structure contains payload data for an Upgrade End Request issued by a client to terminate/close an image download from the server:

```
typedef struct
{
    uint32 u32FileVersion;
    uint16 u16ImageType;
    uint16 u16ManufacturerCode;
           uint8  u8Status;
}tsOTA_UpgradeEndRequestPayload;
```

where:

- u32FileVersion is the file version of the upgrade image which has been downloaded
- u16ImageType is the type of the upgrade image which has been downloaded
- u16ManufacturerCode is the ZigBee-assigned number identifying the manufacturer of the client device from which the request originates
- u8Status is the reported status of the image download, one of:
  - OTA_STATUS_SUCCESS (successfully downloaded and verified)
  - OTA_STATUS_INVALID_IMAGE (downloaded but failed verification)
  - OTA_REQUIRE_MORE_IMAGE (other images needed)
  - OTA_STATUS_ABORT (download in progress is to be aborted)

49.11.12 tsOTA_UpgradeEndResponsePayload

The following structure contains payload data for an Upgrade End Response issued by the server (as the result of an Upgrade End Request from a client):

```
typedef struct
{
    uint32 u32UpgradeTime;
    uint32 u32CurrentTime;
    uint32 u32FileVersion;
    uint16 u16ImageType;
    uint16 u16ManufacturerCode;
}tsOTA_UpgradeEndResponsePayload;
```

where:

- u32UpgradeTime is the UTC time, in seconds, at which the client should upgrade the running image with the downloaded image. If the server does not support UTC time (indicated by a zero value for u32CurrentTime), the client should interpret this value as a time delay before performing the image upgrade
- u32CurrentTime is the current UTC time, in seconds, on the server. If UTC time is not supported by the server, this value should be set to zero. If this value is set to 0xFFFFFFFF, this indicates that the client should wait for an upgrade command from the server before performing the image upgrade

Note: *If the client does not support UTC time but both of the above time values are non-zero, the client will take the difference between the two times as a time delay before performing the image upgrade.*

- u32FileVersion is the file version of the downloaded application image (a wild card value of 0xFFFFFFFF can be used when the same response is sent to client devices from different manufacturers)
- u16ImageType is the type of the downloaded application image (a wild card value of 0xFFFF can be used when the same response is sent to client devices from different manufacturers)
- u16ManufacturerCode is the manufacturer code that was received from the client in the Upgrade End Request message (a wild card value of 0xFFFF can be used when the same response is sent to client devices from different manufacturers)

49.11.13 tsOTA_SuccessBlockResponsePayload

The following structure contains payload data for an Image Block Response which reports ‘success’ and therefore contains a block of image data (see Section 49.11.10):

```
typedef struct
{
    uint8* pu8Data;
            uint32 u32FileOffset;
            uint32 u32FileVersion;
            uint16 u16ImageType;
            uint16 u16ManufacturerCode;
    uint8  u8DataSize;
}tsOTA_SuccessBlockResponsePayload;
```

where:

- pu8Data is a pointer to the start of the data block being transferred
- u32FileOffset is the offset, in bytes, of the start of the data block from the start of the image (normally, the same offset as specified in the Image Block Request)
- u32FileVersion is the file version of the upgrade image to which the included data block belongs
- u16ImageType is the type of the upgrade image to which the included data block belongs
- u16ManufacturerCode is the manufacturer code that was received from the client in the Image Block Request
- u8DataSize is the length, in bytes, of the included data block (this must be less than or equal to the maximum data block length for the client, specified in the Image Block Request)

49.11.14 tsOTA_BlockResponseEvent

The following structure contains payload data for an Image Block Response containing data other than upgrade image data.

```
typedef struct
{
    uint8    u8Status;
    uint8   *pu8Data;
```

```
    uint8    u8DataSize;
}tsOTA_BlockResponseEvent;
```

where:

- u8Status indicates whether a suitable upgrade image is available:
  - OTA_STATUS_SUCCESS: A suitable image is available
  - OTA_STATUS_NO_IMAGE_AVAILABLE: No suitable image is available
- pu8Data is a pointer to the start of the data block being transferred
- u8DataSize is the length, in bytes, of the included data block (this must be less than or equal to the maximum data block length for the client, specified in the Image Block Request)

49.11.15 tsOTA_WaitForData

The following structure contains time information for an Image Block Response. It can be used by a response which reports 'failure', to instruct the client to re-request the data block after a certain waiting time (see Section 49.11.10). It can also be used in 'rate limiting' to specify a new value for the 'block request delay' attribute on the client.

```
typedef struct
{
    uint32 u32CurrentTime;
    uint32 u32RequestTime;
    uint16 u16BlockRequestDelayMs;
}tsOTA_WaitForData;
```

where:

- u32CurrentTime is the current UTC time, in seconds, on the server. If UTC time is not supported by the server, this value should be set to zero
- u32RequestTime is the UTC time, in seconds, at which the client should re-issue an Image Block Request. If the server does not support UTC time (indicated by a zero value for u32CurrentTime), the client should interpret this value as a time delay before re-issuing an Image Block Request

Note: *If the client does not support UTC time but both of the above values are non-zero, the client will take the difference between the two times as a time delay before re-issuing an Image Block Request.*

- u16BlockRequestDelayMs is used in 'rate limiting' to specify the value of the 'block request delay' attribute for the client - this is minimum time, in milliseconds, that the client must wait between consecutive block requests (the client will update the local attribute with this value). If the server does not support rate limiting or does not need to limit the download rate to the client, this field must be set to 0

49.11.16 tsOTA_WaitForDataParams

The following structure is used in the tsOTA_CallBackMessage structure (see Section 49.11.21) on an OTA Upgrade server. It contains the data needed to notify a client that rate limiting is required or the client must wait to receive an upgrade image.

```
typedef struct
{
   bool_t             bInitialized;
   uint16             u16ClientAddress;
   tsOTA_WaitForData  sWaitForDataPyld;
}tsOTA_WaitForDataParams;
```

where:

- `bInitialized` is a boolean flag indicating the server's request to the client:
  TRUE - Implement rate limiting or wait to receive upgrade image
  FALSE - Otherwise
- `u16ClientAddress` contains the 16-bit network address of the client
- `sWaitForDataPyld` is a structure containing the payload for an Image Block Response with status OTA_STATUS_WAIT_FOR_DATA (see Section 49.11.15)

49.11.17 tsOTA_PageReqServerParams

The following structure is used in the `tsOTA_CallBackMessage` structure (see Section 49.11.21) on an OTA Upgrade server. It contains the data from an Image Page Request received from a client.

```
typedef struct
{
    uint8                       u8TransactionNumber;
    bool_t                      bPageReqRespSpacing;
    uint16                      u16DataSent;
    tsOTA_ImagePageRequest      sPageReq;
    tsZCL_ReceiveEventAddress   sReceiveEventAddress;
}tsOTA_PageReqServerParams;
```

where:

- `u8TransactionNumber` is the Transaction Sequence Number (TSN) which is used in the Image Page Request
- `bPageReqRespSpacing` is a boolean used to request a spacing between consecutive Image Block Responses:
  TRUE - Implement spacing
  FALSE - Otherwise
- `u16DataSent` indicates the number of data bytes contained in the Image Page Request
- `sPageReq` is a structure containing the payload data from the Image Page Request (see Section 49.11.9)
- `sReceiveEventAddress` contains the address of the OTA Upgrade client that made the page request

49.11.18 tsOTA_PersistedData

The following structure contains the persisted data that is stored in Flash memory using the NVM module:

```
typedef struct
{
    tsCLD_AS_Ota sAttributes;
    uint32 u32FunctionPointer;
    uint32 u32RequestBlockRequestTime;
    uint32 u32CurrentFlashOffset;
    uint32 u32TagDataWritten;
    uint32 u32Step;
    uint16 u16ServerShortAddress;
#ifdef OTA_CLD_ATTR_REQUEST_DELAY
    bool_t bWaitForBlockReq;
#endif
    uint8 u8ActiveTag[OTA_TAG_HEADER_SIZE];
    uint8 u8PassiveTag[OTA_TAG_HEADER_SIZE];
    uint8 au8Header[OTA_MAX_HEADER_SIZE];
    uint8 u8Retry;
    uint8 u8RequestTransSeqNo;
    uint8 u8DstEndpoint;
```

```
    bool_t bIsCoProcessorImage;
    bool_t bIsSpecificFile;
    bool_t bIsNullImage;
    uint8  u8CoProcessorOTAHeaderIndex;
    uint32 u32CoProcessorImageSize;
    uint32 u32SpecificFileSize;
#ifdef OTA_PAGE_REQUEST_SUPPORT
    tsOTA_PageReqParams sPageReqParams;
#endif
#if (OTA_MAX_CO_PROCESSOR_IMAGES != 0)
    uint8 u8NumOfDownloadableImages;
#endif
#ifdef OTA_INTERNAL_STORAGE
    uint8 u8Buf[4];
#endif
}tsOTA_PersistedData;
```

The fields are for internal use and no knowledge of them is required.

49.11.19 tsOTA_QuerySpecificFileRequestPayload

The following structure contains the payload for a Query Specific File Request which is issued by an OTA Upgrade client to request a device-specific file from the server.

```
typedef struct
{
    uint64 u64RequestNodeAddress;
    uint16 u16ManufacturerCode;
    uint16 u16ImageType;
    uint32 u32FileVersion;
    uint16 u16CurrentZibgeeStackVersion;
}tsOTA_QuerySpecificFileRequestPayload;
```

where:

- `u64RequestNodeAddress` is the IEEE/MAC address of the node requesting the device-specific file from the server
- `u16ManufactuerCode` is the ZigBee-assigned manufacturer code of the requesting node (0xFFFF is used to indicate any manufacturer)
- `u16ImageType` indicates the requested file type - one of the reserved values that are assigned to the device-specific file types (the value should be in the range 0xFFC0 to 0xFFFE, but only 0xFFC0 to 0xFFC2 are currently in use)
- `32FileVersion` contains the release and build numbers of the application and stack that correspond to the device-specific file - for details of the format, refer to the *ZigBee Over-the-Air Upgrading Cluster Specification (095264)*
- `u16CurrentZigbeeStackVersion` contains the version of ZigBee stack that is currently running on the client

49.11.20 tsOTA_QuerySpecificFileResponsePayload

The following structure contains the payload for a Query Specific File Response which is issued by an OTA Upgrade server in response to a request for a device-specific file.

```
typedef struct
{
    uint32 u32FileVersion;
```

```
    uint32 u32ImageSize;
    uint16 u16ImageType;
    uint16 u16ManufacturerCode;
    uint8 u8Status;
}tsOTA_QuerySpecificFileResponsePayload;
```

where:

- `32FileVersion` contains the release and build numbers of the application and stack that correspond to the device-specific file - this field will take the same value as the equivalent field in the corresponding Query Specific File Request (see Section 49.11.19)
- `u32ImageSize` is the size of the requested file, in bytes
- `u16ImageType` indicates the requested file type - this field will take the same value as the equivalent field in the corresponding Query Specific File Request (see Section 49.11.19)
- `u16ManufactuerCode` is the ZigBee-assigned manufacturer code of the requesting node - this field will take the same value as the equivalent field in the corresponding Query Specific File Request (see Section 49.11.19)
- `u8Status` indicates whether a suitable file is available for download:
  - OTA_STATUS_SUCCESS: A suitable file is available
  - OTA_STATUS_NO_IMAGE_AVAILABLE: No suitable file is available

  The other elements of the structure are only included in the case of success.

49.11.21 tsOTA_CallBackMessage

For an OTA 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 `tsOTA_CallBackMessage` structure:

```
typedef struct
{
    teOTA_UpgradeClusterEvents     eEventId;
#ifdef OTA_CLIENT
    tsOTA_PersistedData sPersistedData;
    uint8 au8ReadOTAData[OTA_MAX_BLOCK_SIZE];
    uint8 u8NextFreeImageLocation;
    uint8 u8CurrentActiveImageLocation;
#endif
#ifdef OTA_SERVER
    tsCLD_PR_Ota aServerPrams[OTA_MAX_IMAGES_PER_ENDPOINT
+OTA_MAX_CO_PROCESSOR_IMAGES];
    tsOTA_AuthorisationStruct               sAuthStruct;
    uint8  u8ServerImageStartSector;
    bool bIsOTAHeaderCopied;
    uint8 au8ServerOTAHeader[OTA_MAX_HEADER_SIZE+OTA_TAG_HEADER_SIZE];
    tsOTA_WaitForDataParams  sWaitForDataParams;
#ifdef OTA_PAGE_REQUEST_SUPPORT
    tsOTA_PageReqServerParams  sPageReqServerParams;
#endif
#endif
    uint8  u8ImageStartSector[OTA_MAX_IMAGES_PER_ENDPOINT];
    uint8 au8CAPublicKey[22];
    uint8 u8MaxNumberOfSectors;
    union
    {
        tsOTA_ImageNotifyCommand              sImageNotifyPayload;
        tsOTA_QueryImageRequest               sQueryImagePayload;
```

```
        tsOTA_QueryImageResponse                sQueryImageResponsePayload;
        tsOTA_BlockRequest                      sBlockRequestPayload;
        tsOTA_ImagePageRequest                  sImagePageRequestPayload;
        tsOTA_ImageBlockResponsePayload         sImageBlockResponsePayload;
        tsOTA_UpgradeEndRequestPayload          sUpgradeEndRequestPayload;
        tsOTA_UpgradeEndResponsePayload         sUpgradeResponsePayload;
        tsOTA_QuerySpecificFileRequestPayload sQuerySpFileRequestPayload;
        tsOTA_QuerySpecificFileResponsePayload
                sQuerySpFileResponsePayload;
        teZCL_Status                            eQueryNextImgRspErrStatus;
        tsOTA_SignerMacVerify                   sSignerMacVerify;
        tsOTA_ImageVersionVerify                sImageVersionVerify;
        tsOTA_UpgradeDowngradeVerify            sUpgradeDowngradeVerify;
    }uMessage;
}tsOTA_CallBackMessage;
```

where:

- `eEventId` is the OTA event type (enumerations are detailed in Section 49.12.2)
- `sPersistedData` is the structure (see Section 49.11.18) which contains the persisted data that is stored in Flash memory using the NVM module on the client
- `au8ReadOTAData` is an array containing the payload data from an Image Block Response
- `u8NextFreeImageLocation` identifies the next free image location where a new upgrade image can be stored
- `u8CurrentActiveImageLocation` identifies the location of the currently active image on the client
- `aServerPrams` is an array containing the server data for each image which can be updated by application
- `sAuthStruct` is a structure which stores the authorisation state and list of client devices that are authorised for OTA upgrade
- `u8ServerImageStartSector` identifies the server self-image start-sector
- `bIsOTAHeaderCopied` specifies whether the new OTA header is copied (TRUE) or not (FALSE)
- `au8ServerOTAHeader` specifies the current server OTA header
- `sWaitForDataParams` is a structure containing time information that may need to be modified by the server for inclusion in an Image Block Response (for more information, refer to Section 49.11.15)
- `sPageReqServerParams` is a structure containing page request information that may need to be modified by the server
- `u8ImageStartSector` is used to store the image start-sector for each image which is stored or will be stored in the devices internal Flash memory - note that this variable assumes a 32-Kbyte sector size and so, for example, if
  64-Kbyte sectors are used, its value will be twice the actual start-sector value
- `au8CAPublicKey` specifies the CA public key
- `u8MaxNumberOfSectors` specifies the maximum number of sectors to be used per image
- `uMessage` is a union containing the command payload in one of the following forms (depending on the command specified by `eEventId`):
  - `sImageNotifyPayload` is a structure containing the payload of an Image Notify command
  - `sQueryImagePayload` is a structure containing the payload of a Query Next Image Request
  - `sQueryImageResponsePayload` is a structure containing the payload of a Query Next Image Response
  - `sBlockRequestPayload` is a structure containing the payload of an Image Block Request
  - `sImagePageRequestPayload` is a structure containing the payload of an Image Page Request
  - `sImageBlockResponsePayload` is a structure containing the payload of an Image Block Response
  - `sUpgradeEndRequestPayload` is a structure containing the payload of an Upgrade End Request

- sUpgradeResponsePayload is a structure containing the payload of an Upgrade End Response
- sQuerySpFileRequestPayload is a structure containing the payload of a Query Specific File Request
- sQuerySpFileResponsePayload is a structure containing the payload of a Query Specific File Response
- eQueryNextImgRspErrStatus is the status returned from the query image response command handler and can be passed up to the application when there is an error via the callback event E_CLD_OTA_COMMAND_QUERY_NEXT_IMAGE_RESPONSE_ERROR. The returned status value will be either E_ZCL_ERR_INVALID_IMAGE_SIZE or E_ZCL_ERR_INVALID_IMAGE_VERSION
- sSignerMacVerify is a structure containing the signer's IEEE/MAC address from a new upgrade image and a status field (which is set by the application after verifying the signer's address)
- sImageVersionVerify is a structure containing the image version received in the query next image response and status field (which is set by the application after verifying the image version)
- sUpgradeDowngradeVerify is a structure containing the image version received in the upgrade end response and a status field (which is set by the application after verifying the image version)

49.11.22 tsCLD_PR_Ota

The following structure contains server parameter data that can be pre-set using the function **eOTA_SetServerParams()** and obtained using **eOTA_GetServerData()**:

```
typedef struct
{
    uint8* pu8Data;
    uint32 u32CurrentTime;
    uint32 u32RequestOrUpgradeTime;
    uint8  u8QueryJitter;
    uint8  u8DataSize;
} tsCLD_PR_Ota;
```

where:

- pu8Data is a pointer to the start of a block of data
- u32CurrentTime is the current UTC time, in seconds, on the server. If UTC time is not supported by the server, this value should be set to zero
- u32RequestOrUpgradeTime is used by the server as the 'request time' and the 'upgrade time' when sending responses to clients:
  - As a 'request time', the value may be included in an Image Block Response (see Section 49.11.10 and Section 49.11.15)
  - As an 'upgrade time', the value will be included in an Upgrade End Response (see Section 49.11.12)
- u8QueryJitter is a value between 1 and 100 (inclusive) which is used by a receiving client to decide whether to reply to an Image Notify message - for information on 'Query Jitter', refer to Section 49.7
- u8DataSize is the length, in bytes, of the data block pointed to by pu8Data

49.11.23 tsCLD_AS_Ota

This structure contains attribute values which are stored as part of the persisted data in Flash memory:

```
typedef struct
{
    uint64 u64UgradeServerID;
    uint32 u32FileOffset;
```

```
    uint32 u32CurrentFileVersion;
    uint16 u16CurrentStackVersion;
    uint32 u32DownloadedFileVersion;
    uint16 u16DownloadedStackVersion;
    uint8  u8ImageUpgradeStatus;
    uint16 u16ManfId;
    uint16 u16ImageType;
    uint16 u16MinBlockRequestDelay;
} tsCLD_AS_Ota;
where the structure elements are OTA Upgrade cluster attribute values, as described in Sec
```

49.11.24 tsOTA_ImageVersionVerify

The following structure contains the data for an event of the type E_CLD_OTA_INTERNAL_COMMAND_VERIFY_IMAGE_VERSION.

```
typedef struct
{
    uint32 u32NotifiedImageVersion;
    uint32 u32CurrentImageVersion;
    teZCL_Status eImageVersionVerifyStatus;
}tsOTA_ImageVersionVerify;
```

where:

- `u32NotifiedImageVersion` is the version received in the query next image response
- `u32CurrentImageVersion` is the version of the running image
- `eImageVersionVerifyStatus` is a status field which should be updated to E_ZCL_SUCCESS or E_ZCL_FAIL by the application after checking the received image version, to indicate whether the upgrade image has a valid image version

49.11.25 tsOTA_UpgradeDowngradeVerify

The following structure contains the data for an event of the type E_CLD_OTA_INTERNAL_COMMAND_SWITCH_TO_UPGRADE_DOWNGRADE.

```
typedef struct
{
    uint32 u32DownloadImageVersion;
    uint32 u32CurrentImageVersion;
    teZCL_Status eUpgradeDowngradeStatus;
}tsOTA_UpgradeDowngradeVerify;
```

where:

- `u32DownloadImageVersion` is the version received in upgrade end response
- `u32CurrentImageVersion` is the version of running image
- `eImageVersionVerifyStatus` is a status field which should be updated to E_ZCL_SUCCESS or E_ZCL_FAIL by the application after checking the received image version, to indicate whether the upgrade image has a valid image version

49.12 Enumerations

49.12.1 teOTA_Cluster

The following enumerations represent the OTA Upgrade cluster attributes:

```
typedef enum
{
    E_CLD_OTA_ATTR_UPGRADE_SERVER_ID,
    E_CLD_OTA_ATTR_FILE_OFFSET,
    E_CLD_OTA_ATTR_CURRENT_FILE_VERSION,
    E_CLD_OTA_ATTR_CURRENT_ZIGBEE_STACK_VERSION,
    E_CLD_OTA_ATTR_DOWNLOADED_FILE_VERSION,
    E_CLD_OTA_ATTR_DOWNLOADED_ZIGBEE_STACK_VERSION,
    E_CLD_OTA_ATTR_IMAGE_UPGRADE_STATUS,
    E_CLD_OTA_ATTR_MANF_ID,
    E_CLD_OTA_ATTR_IMAGE_TYPE,
    E_CLD_OTA_ATTR_REQUEST_DELAY
}teOTA_Cluster;
```

The above enumerations are described in the table below.

Table 120. OTA Upgrade Cluster Attributes

Enumeration	Attribute
E_CLD_OTA_ATTR_UPGRADE_SERVER_ID	Upgrade Server ID
E_CLD_OTA_ATTR_FILE_OFFSET	File Offset
E_CLD_OTA_ATTR_CURRENT_FILE_VERSION	Current File Version
E_CLD_OTA_ATTR_CURRENT_ZIGBEE_STACK_VERSION	Current ZigBee Stack Version
E_CLD_OTA_ATTR_DOWNLOADED_FILE_VERSION	Downloaded File Version
E_CLD_OTA_ATTR_DOWNLOADED_ZIGBEE_STACK_VERSION	Downloaded ZigBee Stack Version
E_CLD_OTA_ATTR_IMAGE_UPGRADE_STATUS	Image Upgrade Status
E_CLD_OTA_ATTR_MANF_ID	Manufacturer ID
E_CLD_OTA_ATTR_IMAGE_TYPE	Image Type
E_CLD_OTA_ATTR_REQUEST_DELAY	Minimum Block Request Delay

The above attributes are described in Section 49.3.

49.12.2 teOTA_UpgradeClusterEvents

The following enumerations represent the OTA Upgrade cluster events:

```
typedef enum
{
    E_CLD_OTA_COMMAND_IMAGE_NOTIFY,
    E_CLD_OTA_COMMAND_QUERY_NEXT_IMAGE_REQUEST,
    E_CLD_OTA_COMMAND_QUERY_NEXT_IMAGE_RESPONSE,
    E_CLD_OTA_COMMAND_BLOCK_REQUEST,
    E_CLD_OTA_COMMAND_PAGE_REQUEST,
    E_CLD_OTA_COMMAND_BLOCK_RESPONSE,
    E_CLD_OTA_COMMAND_UPGRADE_END_REQUEST,
    E_CLD_OTA_COMMAND_UPGRADE_END_RESPONSE,
    E_CLD_OTA_COMMAND_QUERY_SPECIFIC_FILE_REQUEST,
```

```
    E_CLD_OTA_COMMAND_QUERY_SPECIFIC_FILE_RESPONSE,
    E_CLD_OTA_INTERNAL_COMMAND_TIMER_EXPIRED,
    E_CLD_OTA_INTERNAL_COMMAND_SAVE_CONTEXT,
    E_CLD_OTA_INTERNAL_COMMAND_OTA_DL_ABORTED,
    E_CLD_OTA_INTERNAL_COMMAND_POLL_REQUIRED,
    E_CLD_OTA_INTERNAL_COMMAND_RESET_TO_UPGRADE,
    E_CLD_OTA_INTERNAL_COMMAND_LOCK_FLASH_MUTEX,
    E_CLD_OTA_INTERNAL_COMMAND_FREE_FLASH_MUTEX,
    E_CLD_OTA_INTERNAL_COMMAND_SEND_UPGRADE_END_RESPONSE,
    E_CLD_OTA_INTERNAL_COMMAND_CO_PROCESSOR_BLOCK_RESPONSE,
    E_CLD_OTA_INTERNAL_COMMAND_CO_PROCESSOR_DL_ABORT,
    E_CLD_OTA_INTERNAL_COMMAND_CO_PROCESSOR_IMAGE_DL_COMPLETE,
    E_CLD_OTA_INTERNAL_COMMAND_CO_PROCESSOR_SWITCH_TO_NEW_IMAGE,
    E_CLD_OTA_INTERNAL_COMMAND_CO_PROCESSOR_IMAGE_BLOCK_REQUEST,
    E_CLD_OTA_INTERNAL_COMMAND_SPECIFIC_FILE_BLOCK_RESPONSE,
    E_CLD_OTA_INTERNAL_COMMAND_SPECIFIC_FILE_DL_ABORT,
    E_CLD_OTA_INTERNAL_COMMAND_SPECIFIC_FILE_DL_COMPLETE,
    E_CLD_OTA_INTERNAL_COMMAND_SPECIFIC_FILE_USE_NEW_FILE,
    E_CLD_OTA_INTERNAL_COMMAND_SPECIFIC_FILE_NO_UPGRADE_END_RESPONSE,
    E_CLD_OTA_COMMAND_QUERY_NEXT_IMAGE_RESPONSE_ERROR,
    E_CLD_OTA_INTERNAL_COMMAND_VERIFY_SIGNER_ADDRESS,
    E_CLD_OTA_INTERNAL_COMMAND_RCVD_DEFAULT_RESPONSE,
    E_CLD_OTA_INTERNAL_COMMAND_VERIFY_IMAGE_VERSION,
    E_CLD_OTA_INTERNAL_COMMAND_SWITCH_TO_UPGRADE_DOWNGRADE,
    E_CLD_OTA_INTERNAL_COMMAND_REQUEST_QUERY_NEXT_IMAGES,
    E_CLD_OTA_INTERNAL_COMMAND_OTA_START_IMAGE_VERIFICATION_IN_LOW_PRIORITY,
    E_CLD_OTA_INTERNAL_COMMAND_FAILED_VALIDATING_UPGRADE_IMAGE,
    E_CLD_OTA_INTERNAL_COMMAND_FAILED_COPYING_SERIALIZATION_DATA
}teOTA_UpgradeClusterEvents;
```

The above enumerations are described in the table below.

Table 121. OTA Upgrade Cluster Events

Enumeration	Event Description
E_CLD_OTA_COMMAND_IMAGE_NOTIFY	Generated on client when an Image Notify message is received from the server to indicate that a new application image is available for download
E_CLD_OTA_COMMAND_QUERY_NEXT_IMAGE_REQUEST	Generated on server when a Query Next Image Request is received from a client to enquire whether a new application image is available for download
E_CLD_OTA_COMMAND_QUERY_NEXT_IMAGE_RESPONSE	Generated on client when a Query Next Image Response is received from the server (in response to a Query Next Image Request) to indicate whether a new application image is available for download
E_CLD_OTA_COMMAND_BLOCK_REQUEST	Generated on server when an Image Block Request is received from a client to request a block of image data as part of a download
E_CLD_OTA_COMMAND_PAGE_REQUEST	Generated on server when an Image Page Request is received from a client to request a page of image data as part of a download
E_CLD_OTA_COMMAND_BLOCK_RESPONSE	Generated on client when an Image Block Response is received from the server (in response to an Image Block Request) and contains a block of image data which is part of a download

Table 121. OTA Upgrade Cluster Events*...continued*

Enumeration	Event Description
E_CLD_OTA_COMMAND_UPGRADE_END_REQUEST	Generated on server when an Upgrade End Request is received from a client to indicate that the complete image has been downloaded and verified
E_CLD_OTA_COMMAND_UPGRADE_END_RESPONSE	Generated on client when an Upgrade End Response is received from the server (in response to an Upgrade End Request) to confirm the end of a download
E_CLD_OTA_COMMAND_QUERY_SPECIFIC_FILE_ REQUEST	Generated on server when a Query Specific File Request is received from a client to request a particular application image
E_CLD_OTA_COMMAND_QUERY_SPECIFIC_FILE_ RESPONSE	Generated on client when a Query Specific File Response is received from the server (in response to a Query Specific File Request) to indicate whether the requested application image is available for download
E_CLD_OTA_INTERNAL_COMMAND_TIMER_EXPIRED	Generated on client to notify the application that the local one-second timer has expired
E_CLD_OTA_INTERNAL_COMMAND_SAVE_CONTEXT	Generated on server or client to prompt the application to store context data in Flash memory
E_CLD_OTA_INTERNAL_COMMAND_OTA_DL_ABORTED	Generated on a client if the received image is invalid or the client has aborted the image download (allowing the application to request the new image again)
E_CLD_OTA_INTERNAL_COMMAND_POLL_REQUIRED	Generated on client to prompt the application to poll the server for a new application image
E_CLD_OTA_INTERNAL_COMMAND_RESET_TO_UP GRADE	Generated on client to notify the application that the stack is going to reset the device
E_CLD_OTA_INTERNAL_COMMAND_LOCK_-FLASH_ MUTEX	Generated on server or client to prompt the application to lock the mutex used for accesses to external Flash memory
E_CLD_OTA_INTERNAL_COMMAND_FREE_-FLASH_ MUTEX	Generated on server or client to prompt the application to unlock the mutex used for accesses to external Flash memory
E_CLD_OTA_INTERNAL_COMMAND_SEND_UPGRADE_ END_RESPONSE	Generated on server to notify the application that the stack is going to send an Upgrade End Response to a client
E_CLD_OTA_INTERNAL_COMMAND_CO_PROCESSOR_ BLOCK_RESPONSE	Generated on client to notify the application that Image Block Response has been received for co-processor image
E_CLD_OTA_INTERNAL_COMMAND_CO_PROCESSOR_ DL_ABORT	Generated on client to notify the application that download of co-processor image from the server has been aborted
E_CLD_OTA_INTERNAL_COMMAND_CO_PROCESSOR_ IMAGE_DL_COMPLETE	Generated on client to notify the application that download of co-processor image from the server has completed
E_CLD_OTA_INTERNAL_COMMAND_CO_PROCESSOR_ SWITCH_TO_NEW_IMAGE	Generated on client to notify the application that the upgrade time for a previously downloaded co-processor image has been reached (this event is generated after receiving the Upgrade End Response which contains the upgrade time)
E_CLD_OTA_INTERNAL_COMMAND_CO_PROCESSOR_ IMAGE_BLOCK_REQUEST	Generated on server when an Image Block Request is received from a client to request a block of image data

Table 121. OTA Upgrade Cluster Events*...continued*

Enumeration	Event Description
	as part of a download and the server finds that the required image is stored in the co-processor's external storage device
E_CLD_OTA_INTERNAL_COMMAND_SPECIFIC_FILE_BLOCK_RESPONSE	Generated on client when an Image Block Response is received from server as part of a device-specific file download - the event contains a block of file data which the client stores in an appropriate location
E_CLD_OTA_INTERNAL_COMMAND_SPECIFIC_FILE_DL_ABORT	Generated on client when the final Image Block Response of a device-specific file download has been received from the server
E_CLD_OTA_INTERNAL_COMMAND_SPECIFIC_FILE_DL_COMPLETE	Generated on client following a device-specific file download to indicate that the upgrade time has been reached and the file can now be used by the client
E_CLD_OTA_INTERNAL_COMMAND_SPECIFIC_FILE_USE_NEW_FILE	Generated to indicate that a device-specific file download is being aborted and any received data must be discarded by the application
E_CLD_OTA_INTERNAL_COMMAND_SPECIFIC_FILE_NO_UPGRADE_END_RESPONSE	Generated to indicate that no Upgrade End Response has been received for a device-specific file download (after three attempts to obtain one)
E_CLD_OTA_COMMAND_QUERY_NEXT_IMAGE_RESPONSE_ERROR	This event is generated on the client when a Query Next Image Response message is received from the server, in response to a Query Next Image Request with a status of Invalid Image Size.
E_CLD_OTA_INTERNAL_COMMAND_VERIFY_SIGNER_ADDRESS	This event is generated to prompt the application to verify the signer address received in a new OTA upgrade image. This event gives control to the application to verify that the new upgrade image came from a trusted source. After checking the signer address, the application should set the status field of the event to E_ZCL_SUCCESS (valid source) or E_ZCL_FAIL (invalid source).
E_CLD_OTA_INTERNAL_COMMAND_RCVD_DEFAULT_RESPONSE	This event is generated on the client when a default response message is received from the server, in response to a Query Next Image Request, Image Block Request or Upgrade End Request. This is an internal ZCL event that results in an OTA download being aborted, thus activating the callback function for the event E_CLD_OTA_INTERNAL_COMMAND_OTA_DL_ABORTED.
E_CLD_OTA_INTERNAL_COMMAND_VERIFY_IMAGE_VERSION	This event is generated to prompt the application to verify the image version received in a Query Next Image Response. This event allows the application to verify that the new upgrade image has a valid image version. After checking the image version, the application should set the status field of the event to E_ZCL_SUCCESS (valid version) or E_ZCL_FAIL (invalid version).
E_CLD_OTA_INTERNAL_COMMAND_SWITCH_TO_UPGRADE_DOWNGRADE	This event is generated to prompt the application to verify the image version received in an upgrade end response. This event allows the application to verify that the new upgrade image has a valid image version. After checking the image version, the application should set the status field of the event to E_ZCL_SUCCESS (valid version) or E_ZCL_FAIL (invalid version).

Table 121. OTA Upgrade Cluster Events*...continued*

Enumeration	Event Description
E_CLD_OTA_INTERNAL_COMMAND_REQUEST_QUERY_NEXT_IMAGES	This event is generated on the client when a co-processor image also requires the client to update its own image. After the first file is downloaded (co-processor image) this event notifies the application to allow it to send a Query Next Image command for its own upgrade image, using the function **eOTA_ClientQueryNextImageRequest()**.
E_CLD_OTA_INTERNAL_COMMAND_OTA_START_IMAGE_VERIFICATION_IN_LOW_PRIORITY	This event is generated to prompt the application to verify the downloaded JN516x client image from a low priority task. Once the low priority task is running, the application should call **eOTA_VerifyImage()** to start image verification.
E_CLD_OTA_INTERNAL_COMMAND_FAILED_VALIDATING_UPGRADE_IMAGE	This event is generated on the client when the validation of a new upgrade image fails. This validation takes place when the upgrade time is reached.
E_CLD_OTA_INTERNAL_COMMAND_FAILED_COPYING_SERIALIZATION_DATA	This event is generated on the client when the copying of serialization data from the active image to the new upgrade image fails. This process takes place after image validation (if applicable) are completed successfully.

The above events are described in more detail in Section 49.9.

49.12.3 eOTA_AuthorisationState

The following enumerations represent the authorisation options concerning which clients are allowed to obtain upgrade images from the server:

```
typedef enum
{
    E_CLD_OTA_STATE_ALLOW_ALL,
    E_CLD_OTA_STATE_USE_LIST
}eOTA_AuthorisationState;
```

The above enumerations are described in the table below.

Table 122. Client Authorisation Options

Enumeration	Description
E_CLD_OTA_STATE_ALLOW_ALL	Allow all clients to obtain upgrade images from this server
E_CLD_OTA_STATE_USE_LIST	Only allow clients in authorisation list to obtain upgrade images from this server

49.12.4 teOTA_ImageNotifyPayloadType

The following enumerations represent the payload options for an Image Notify message issued by the server:

```
typedef enum
{
    E_CLD_OTA_QUERY_JITTER,
    E_CLD_OTA_MANUFACTURER_ID_AND_JITTER,
    E_CLD_OTA_ITYPE_MDID_JITTER,
    E_CLD_OTA_ITYPE_MDID_FVERSION_JITTER
}teOTA_ImageNotifyPayloadType;
```

The above enumerations are described in the table below.

Table 123. Image Notify Payload Options

Enumeration	Description
E_CLD_OTA_QUERY_JITTER	Include only ‘Query Jitter’ in payload
E_CLD_OTA_MANUFACTURER_ID_AND_JITTER	Include ‘Manufacturer Code’ and ‘Query Jitter’ in payload
E_CLD_OTA_ITYPE_MDID_JITTER	Include ‘Image Type’, ‘Manufacturer Code’ and ‘Query Jit-ter’ in payload
E_CLD_OTA_ITYPE_MDID_FVERSION_JITTER	Include ‘Image Type’, ‘Manufacturer Code’, ‘File Version’ and ‘Query Jitter’ in payload

49.13 Compile-time options

To enable the OTA Upgrade cluster in the code to be built, it is necessary to add the following to the **zcl_options.h** file:

```
#define CLD_OTA
```

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
```

Note: *The OTA Upgrade cluster must be enabled as a client or server, as appropriate, in the application images to be downloaded using the cluster. The relevant cluster options (see below) should also be enabled for the image.*

The following may also be defined in the **zcl_options.h** file.

Optional Attributes (Client only)

The OTA Upgrade cluster has attributes on the client side only. The optional attributes may be specified by defining some or all of the following.

Add this line to enable the optional File Offset attribute:

```
#define OTA_CLD_ATTR_FILE_OFFSET
```

Add this line to enable the optional Current File Version attribute:

```
#define OTA_CLD_ATTR_CURRENT_FILE_VERSION
```

Add this line to enable the optional Current ZigBee Stack Version attribute:

```
#define OTA_CLD_ATTR_CURRENT_ZIGBEE_STACK_VERSION
```

Add this line to enable the optional Downloaded File Version attribute:

```
#define OTA_CLD_ATTR_DOWNLOADED_FILE_VERSION
```

Add this line to enable the optional Downloaded ZigBee Stack Version attribute:

```
#define OTA_CLD_ATTR_DOWNLOADED_ZIGBEE_STACK_VERSION
```

Add this line to enable the optional Manufacturer ID attribute:

```
#define         OTA_CLD_MANF_ID
```

Add this line to enable the optional Image Type attribute:

```
#define OTA_CLD_IMAGE_TYPE
```

Add this line to enable the optional Minimum Block Request Delay attribute:

```
#define OTA_CLD_ATTR_REQUEST_DELAY
```

Global Attributes

Add this line to define the value (n) of the Cluster Revision attribute:

```
#define CLD_THERMOSTAT_CLUSTER_REVISION <n>
```

The default value is 1, which corresponds to the revision of the cluster in the ZCL r6 specification (see Section 2.4).

Number of Images

The maximum number of images that can be stored in the Flash memory of the device of a server or client node must be specified as follows, where in this example the maximum is two images:

```
#define OTA_MAX_IMAGES_PER_ENDPOINT             2
```

Note that the active image should not be included.

OTA Block Size

The maximum size of a block of image data to be transferred over the air is defined, in bytes, as follows:

```
#define OTA_MAX_BLOCK_SIZE                      100
```

If a large maximum block size is configured, it is recommended to enable fragmentation for data transfers between nodes. Fragmentation is enabled and configured on the sending and receiving nodes as described in the ‘Application Design Notes’ appendix of the *ZigBee 3.0 Stack User Guide (JNUG3130)*.

Page Requests

The ‘page request’ feature can be enabled on the server and client by adding the line:

```
#define OTA_PAGE_REQUEST_SUPPORT
```

If the page request feature is enabled then the page size (in bytes) and 'response spacing' (in milliseconds) to be inserted into the Image Page Requests can be configured by defining the following macros on the client:

```
#define OTA_PAGE_REQ_PAGE_SIZE                    512
#define OTA_PAGE_REQ_RESPONSE_SPACING             300
```

The above example definitions contain the default values of 512 bytes and 300 ms.

Hardware Versions in OTA Header

If hardware versions will be present in the OTA header then in order to enable checks of the hardware versions on the OTA server and client, add:

```
#define OTA_CLD_HARDWARE_VERSIONS_PRESENT
```

Custom Serialization Data

To maintain custom serialization data associated with binary images during upgrades on the server or client, add:

```
#define         OTA_MAINTAIN_CUSTOM_SERIALISATION_DATA
```

OTA Command Acks

To disable APS acknowledgements for OTA commands on the server or client, add:

```
 #define         OTA_ACKS_ON   FALSE
```

If the above define is not included, APS acknowledgements are enabled by default. **They must be enabled for ZigBee certification**, but for increased download speed it may be convenient to disable them during application development. However, they must not be disabled if using fragmentation.

Frequency of Requests (Client only)

To avoid flooding the network with continuous packet exchanges, the request messages from the client can be throttled by defining a time interval, in seconds, between consecutive requests. For example, a one-second interval is defined as follows:

```
#define OTA_TIME_INTERVAL_BETWEEN_REQUESTS       1
```

If this time interval is not defined then the time interval, in seconds, between consecutive retries of an unthrottled message request should be defined. For example, a ten-second retry interval is defined as follows:

```
#define OTA_TIME_INTERVAL_BETWEEN_RETRIES        10
```

(valid only if OTA_TIME_INTERVAL_BETWEEN_REQUESTS is not defined)

Upper Limit on Minimum Block Request Delay

An upper limit on the value of the Minimum Block Request Delay attribute is defined, in seconds, as follows:

```
#define OTA_BLOCK_REQUEST_DELAY_MAX_VALUE        2
```

In the above example, the limit is set to 2 seconds. If no value is defined, the default value of this limit is 5 seconds.

Device Address Copying

On a device whose application image is to be upgraded (client or server), the OTA Upgrade cluster must copy the IEEE/MAC address of the device from the old image to the new image. This copy must be enabled on the device by adding the line:

```
#define OTA_COPY_MAC_ADDRESS
```

No Security Certificate

If no security certificate is to be used, it is necessary to remove references to the Certicom security certificate by including the following definition:

```
#define OTA_NO_CERTIFICATE
```

Internal Storage of OTA Upgrade Image on Client

An OTA upgrade image can be stored in the devices internal Flash memory on an OTA Upgrade cluster client by including the following definition:

```
#define OTA_INTERNAL_STORAGE
```

In addition, if the OTA upgrade image is encrypted then it needs to be decrypted before being stored in internal Flash memory. This decryption can be enabled by including the following definition:

```
#define INTERNAL_ENCRYPTED
```

49.14 Build Process

Special build requirements must be implemented when building applications that are to participate in OTA upgrades:

1. Certain lines must be included in the makefiles for the applications - see Section 49.14.1

2. The server and client applications must then be built - see Section 49.14.2

3. The (initial) client application must now be prepared and loaded into Flash memory of the client device - see Section 49.14.3

4. The server application must now be prepared and loaded into Flash memory of the server device - see Section 49.14.4

49.14.1 Modifying Makefiles

In the makefiles for all applications (for server and all clients), replace the following lines:

```
$(OBJCOPY) -j .version -j .bir -j .flashheader -j .vsr_table
-j .vsr_handlers  -j .rodata -j .text -j .data -j .bss -j .heap
-j .stack -S -O binary $< $@
```

with:

```
        $(OBJCOPY) -j .version -j .bir -j .flashheader -j .vsr_table
-j .vsr_handlers -j .ro_mac_address -j .ro_ota_header  -j .rodata
-j .text -j .data -j .bss -j .heap -j .stack -S -O binary $< $@
```

49.14.2 Building Applications

The server and client applications must be built with the makefiles adapted for OTA upgrade (see Section 49.14.1). A build can be conducted from MCUXpresso as for any ZigBee PRO application - refer to the *MCUXpresso Installation and User Guide (JNUG3136)*.

The resulting binary files must then be prepared and loaded into Flash memory as described in Section 49.14.3 and Section 49.14.4.

49.14.3 Preparing and Downloading Initial Client Image

The first time that the client is programmed with an application, the binary image must be loaded into Flash memory on the client device using a Flash programming tool such as the Flash Programmer within MCUXpresso (normally only used in a development environment) or the *DK6 Production Flash Programmer (JN-SW-4407)*.

After this initial image has been loaded, all subsequent client images will be downloaded from the server to the client via the OTA Upgrade cluster.

49.14.4 Preparing and Downloading Server Image

The server device is programmed by loading a binary image into Flash memory using a Flash programming tool such as the Flash Programmer within MCUXpresso (normally only used in a development environment) or the *DK6 Production Flash Programmer (JN-SW-4407)*.

When a new client image becomes available for the server to distribute, this image must be loaded into the server.

- In a deployed and running system, this image may be supplied via a backhaul network.
- In a development environment, it may be loaded into Flash memory using the Flash Programmer within MCUXpresso.

  However, this Flash Programmer only allows programming from the start of Flash memory. Therefore, the server application must be re-programmed into the Flash memory as well as the new client image. The server application binary and client application binary must be combined into a single binary image using the DK6 Encryption Tool (JET) before being loaded into the server. This tool is provided in the SDK and is described in the *JET User Guide (JNUG3135)*.

Note: *If desired, the initial server image can also include the initial client application. Although there is no need for the server to download this first client application to the client(s), it may be stored in the server in case there is any subsequent need to re-load it into a client.*

49.15 OTA Configuration for Internal Flash

OTA cluster is enabled through the **ZCL_options.h** file.

The OTA cluster requires initialization of the location where the upgrade image can be stored. The application provides this through ***eOTA_AllocateEndpointOTASpace*** API.

Each page on the device is 512 bytes. The usable flash size is 632 K, with 32 K typically reserved for NVM (start page 1152) and 24 K for customer data. This leaves a usable flash size for image at 576 K.

If we split it into two sections to support OTA. It means 288 K becomes maximum image size. Each 288 K section would be 576 flash pages.

This could be represented as 32 K sectors to keep in line with legacy devices.

So, for allocation to the OTA cluster:

```
uint8 u8MaxSectorPerImage = 0;
uint8 u8StartSector[1] = {9}; /
* So next image starts at 9*32*1024 = 288K offset*/
u8MaxSectorPerImage = 9 ; /
* 9 *32* 1024 = 288K is the maximum size of the image */
sNvmDefs.u32SectorSize = 512; /* Sector Size = 512 bytes*/
```

The OTA checks for the presence of the well-known Zigbee09 key at a fixed location within the image.

This provides a convenient mechanism to test the decryption of an encrypted image and an additional sanity check to make sure the image is a valid image to progress downloading.

Note: *This is not the mechanism for a full image validation. For a better validation of an OTA image, it is recommended that for OTA an encrypted image with the CRC check is used.*

Each application note has an OTA_BUILD folder which holds the OTA compatible images.

LinkKey_3.txt is required for creation of the OTA image. It holds the Key which can be used for validation purpose as described above.

The **config OTA_JN518x_Cer_Keys_HA_Light.txt** and **config OTA_JN518x_Cer_Keys_HA_Light_Generic.txt** files provide the OTA image generator with the offset for the key.

Prior to this release the values were **LinkKey_3.txt,02c0,16**, which should now be **LinkKey_3.txt,01b0,16**.

49.15.1 Switching to a new image

After an image is programmed into flash and validated as a correct image to switch to, the following steps should then be performed. These are already done as part of the OTA cluster. However, for custom applications not using the OTA cluster, it would be a requirement to switch and run a new image.

```
    /* Offset in bytes into flash for the new image. In this example, new image
is at 288K boundary*/
  u32Offset = 0x48000;
    /* Set the active image location */
    psector_SetEscoreImageData(u32Offset, 0);
    /* Disable any interrupts during the switching of the images */
    __disable_irq();
    /* Remap the vector table to point to the ROM instead of the application */
    SYSCON->MEMORYREMAP &= ~(SYSCON_MEMORYREMAP_MAP_MASK <<
SYSCON_MEMORYREMAP_MAP_SHIFT);
    /* Prevent entering into Low Power Mode when we reset */
    PMC->CTRL &= ~(PMC_CTRL_LPMODE_MASK << PMC_CTRL_LPMODE_SHIFT);
    /* Initiate reset */
    NVIC_SystemReset();
```

Part XIII: Appendices

This part comprises nine appendices covering topics that include mutex callbacks, attribute reporting, attribute discovery, custom endpoints, manufacturer-specific attributes and commands, the storage of OTA upgrade applications in the devices Internal or External Flash memory, the OTA upgrade of nodes comprising two processors, example code fragments and a glossary of terms.

50 Appendix A: Mutex callbacks

The mutexes are designed such that a call to **ZPS_u8GrabMutexLock()** must be followed by a call to **ZPS_u8ReleaseMutexLock()**. In addition, the call must not be followed by another call to **ZPS_u8GrabMutexLock()**, which means the mutexes are binary rather than counting. This can cause problems if the ZCL takes a mutex via the callback function and then the application wants to lock the mutex to access the shared device structures. Some ZCL clusters also invoke the callback function with E_ZCL_CBET_LOCK_MUTEX multiple times. The counting mutex code below should be used in the application code. When the application wants to access the shared structure, it should call the **vLockZCLMutex()** function (shown in the code extract below), rather than **ZPS_u8GrabMutexLock()**, so that it also participates in the counting mutex rather than directly taking the binary ZPS mutex-protection. Similarly, the shared structure should be released using **vUnlockZCLMutex()**.

The code below uses a single resource for all endpoints and the general callback function. It defines a file scope counter that is the mutex count related to the resource.

At the top of the application source file, create the count and lock/unlock mutex function prototypes (these prototypes may be placed in a header file, if desired):

```
uint32  u32ZCLMutexCount  =  0;
void  vLockZCLMutex(void);
void  vUnlockZCLMutex(void);
```

In both **cbZCL_GeneralCallback()** and **cbZCL_EndpointCallback()**, make the calls:

```
switch(psEvent->eEventType)
 {
 case  E_ZCL_CBET_LOCK_MUTEX:
              vLockZCLMutex();
            break;
      case  E_ZCL_CBET_UNLOCK_MUTEX:
        vUnlockZCLMutex();
            break;
```

Define the lock/unlock mutex functions and call them from the application when accessing any ZCL shared structure:

```
void  vLockZCLMutex(void)
{
        if  (u32ZCLMutexCount  ==  0)
        {
                ZPS_u8GrabMutexLock(mutexZCL,0);
        }
        u32ZCLMutexCount++;
}
void  vUnlockZCLMutex(void)
{
        u32ZCLMutexCount--;
        if  (u32ZCLMutexCount  ==  0)
        {
                ZPS_u8ReleaseMutexLock(mutexZCL,0);
        }
}
```

51 Appendix B: Attribute reporting

Attribute reporting involves sending attribute values unsolicited from the cluster server to a client - that is, pushing values from server to client without the client needing to request the values. This mechanism reduces network traffic compared with the client polling the server for attribute values. It also allows a sleeping server to report its attribute values while it is awake.

The server sends an ‘attribute report’ to the client, where this report can be issued in one of the following ways:

- by a function call in the user application (on the server device)
- automatically by the ZCL (triggered by a change in the attribute value or periodically)

The rules for automatic reporting (see Appendix B.1) can be configured by a remote client by sending a ‘configure reporting’ command to the server. The same rules apply to ‘default reporting’ (see Appendix B.2), but are configured locally on the server. The configuration of attribute reporting is described in Appendix B.3. Remote devices can also query the attribute reporting configuration of the server, as described in Appendix B.6.

Sending and receiving attribute reports are described in Appendix B.4 and Appendix B.5.

Attention: *Attribute reporting is an optional feature and is not supported by all devices.*

51.1 Appendix B.1: Automatic attribute reporting

Automatic attribute reporting involves two mechanisms:

- A report is triggered by a change in the attribute value of at least a configured minimum amount
- Reports are issued for the attribute periodically at a configured frequency

These mechanisms can operate at the same time. In this case, reports will be issued periodically and additional reports will be issued between periodic reports if triggered by changes in the attribute value.

If reports are triggered by frequent changes in the attribute value, they may add significantly to the network traffic. To manage this traffic, the production of reports for an attribute can be ‘throttled’. This involves defining a minimum time-interval between consecutive reports for the attribute. If the attribute value changes within this time-interval since the last report, a new report will not be generated.

Note: *If triggered reports are throttled, periodic reports will still be produced as scheduled.*

Periodic reporting can be disabled, leaving only triggered reports to be automatically generated. Automatic reporting can also be disabled altogether (both mechanisms). For information on the configuration of automatic reporting, refer to Appendix B.3.

51.2 Appendix B.2: Default reporting

For each cluster, the ZCL specification states that certain attributes must be reportable. These attributes are specified in the cluster descriptions in this manual. Reports on these attributes are optional and can be enabled on an individual basis using a ‘reportable flag’, as described in Appendix B.3.6. The attributes for which the flag is set will always be reported, defining a set of attributes for ‘default reporting’.

Default reporting is a form of automatic reporting (see Appendix B.1) for the restricted set of attributes described above. It is configured on the cluster server as described in Appendix B.3.6. The attributes enabled for default reporting are also included in attribute reporting initiated by the server application though a call to the function **eZCL_ReportAllAttributes()**.

51.3 Appendix B.3: Configuring attribute reporting

If attribute reporting is to be used by a cluster then the feature must be enabled at compile-time, as detailed in Appendix B.3.1. Then:

- If automatic attribute reporting is to be implemented then the reports must be configured as described in Appendix B.3.5.
- If default reporting is to be implemented then the reports must be configured as described in Appendix B.3.6.

The ZCL configuration for attribute reporting is described in Appendix B.3.7 for users who wish to modify this configuration.

51.3.1 B.3.1: Compile-time Options

Attribute reporting is enabled at compile-time by setting the appropriate macros in **zcl_options.h**. The compile-time options relevant to the cluster server and client are listed separately below.

51.3.2 B.3.2: Server Options

Generate Attribute Reports

To enable a server to generate attribute reports according to configured reporting rules, add the following option:

```
#define ZCL_ATTRIBUTE_REPORTING_SERVER_SUPPORTED
```

Note: *Attribute reporting does not need to be enabled with this macro if the reports are generated only via function calls.*

Handle ‘Configure Reporting’ Commands

To enable a server to handle ‘configure reporting’ commands and reply with ‘configure reporting’ responses, add the following option:

```
#define ZCL_CONFIGURE_ATTRIBUTE_REPORTING_SERVER_SUPPORTED
```

Handle ‘Read Reporting Configuration’ Commands

To enable a server to handle ‘read reporting configuration’ commands and reply with ‘read reporting configuration’ responses, add the following option:

```
#define ZCL_READ_ATTRIBUTE_REPORTING_CONFIGURATION_SERVER_SUPPORTED
```

Number of Attribute Reports

The number of reportable attributes can be set (to n) using the following line:

```
#define ZCL_NUMBER_OF_REPORTS    n
```

The default value is 10.

Number of String Attribute Reports

The number of reportable string attributes can be set (to n) using the following line:

```
#define ZCL_NUMBER_OF_STRING_REPORTS    n
```

The default value is 0 (meaning that string attribute reports are disabled by default).

Maximum Size of Reportable String Attribute

The maximum size, in bytes, of a string attribute that can be reported can be set (to n) using the following line:

```
#define ZCL_ATTRIBUTE_REPORT_STRING_MAXIMUM_SIZE   n
```

The default value is 32 bytes.

Minimum Attribute Reporting Interval

The minimum time-interval, in seconds, between consecutive attribute reports can be set (to n) using the following line:

```
#define ZCL_SYSTEM_MIN_REPORT_INTERVAL   n
```

The default value is 1 second.

Maximum Attribute Reporting Interval

The maximum time-interval, in seconds, between consecutive attribute reports can be set (to n) using the following line:

```
#define ZCL_SYSTEM_MAX_REPORT_INTERVAL   n
```

The default value is 61 seconds.

51.3.3 B.3.3: Client Options

Receive Attribute Reports

To enable a client to receive attribute reports from a server, add the following option:

```
#define ZCL_ATTRIBUTE_REPORTING_CLIENT_SUPPORTED
```

Send ‘Configure Reporting’ Commands

To enable a client to send ‘configure reporting’ commands and handle the ‘configure reporting’ responses, add the following option:

```
#define ZCL_CONFIGURE_ATTRIBUTE_REPORTING_CLIENT_SUPPORTED
```

Send ‘Read Reporting Configuration’ Commands

To enable a client to send ‘read reporting configuration’ commands and handle the ‘read reporting configuration’ responses, add the following option:

```
#define ZCL_READ_ATTRIBUTE_REPORTING_CONFIGURATION_CLIENT_SUPPORTED
```

51.3.4 B.3.4: General (Server and Client) Options

If attribute reporting is to report any attributes of the 'floating point' type, the following macro must also be enabled in **zcl_options.h** on both the server and client:

```
#define ZCL_ENABLE_FLOAT
```

This enables the use of the floating point library to calculate differences in attribute values. If this library is not already used by the application code, enabling it in this way increases the build size of the application by approximately 5 Kbytes.

51.3.5 B.3.5: Configuring Automatic Attribute Reports (from Client)

If automatic attribute reporting is to be employed between a cluster server and client, the reporting rules must be configured. These rules include the following parameters for each attribute:

- Time-interval between consecutive reports in periodic reporting
- Minimum time-interval between consecutive triggered attribute reports
- Minimum change in the attribute value that will trigger an attribute report

***Note:** **Note 1:** Setting the periodic reporting time-interval to the special value of 0x0000 disables periodic reporting for the attribute. Setting this time-interval to the special value of 0xFFFF disables automatic reporting completely (periodic and triggered) for the attribute.*

***Note:** **Note 2:** Automatic attributes reports are normally produced on a timescale of seconds. However, reports generated on the change of an attribute value can be speeded up to occur on a timescale of milliseconds, as described in Appendix B.3.8.*

***Note:** **Note 3:** Before automatic reporting can be configured on an attribute, the 'reportable flag' must be set for the attribute on the cluster server using the function **eZCL_SetReportableFlag()**.*

This configuration is conducted on the cluster server but is normally directed from a remote device via 'configure reporting' commands.

The configuration of automatic attribute reporting follows the process:

1. The client sends a 'configure reporting' command to the server.

2. The server receives and processes the command, configures the attribute reporting and generates a 'configure reporting' response, which it sends back to the requesting client.

3. The client receives the 'configure reporting' response and the ZCL generates events to indicate the status of the request to the client.

These steps are described separately below.

1. Sending a 'Configure Reporting' Command (from Client)

The application on the cluster client device can configure attribute reporting for a set of attributes on the cluster server using the function **eZCL_SendConfigureReportingCommand()**. This function sends a 'configure reporting' command to the server.

In this function call, a pointer must be provided to an array of `tsZCL_AttributeReportingConfigurationRecord` structures, where each structure contains the configuration details for one attribute on which reporting is to be configured (see Section 6.1.5).

2. Receiving a ‘Configure Reporting’ Command (on Server)

The server will automatically process an incoming ‘configure reporting’ command and perform the required configuration without assistance from the application. For each attribute (in the configuration request), the reporting configuration values are parsed, after which the ZCL generates an event of the type:

E_ZCL_CBET_REPORT_INDIVIDUAL_ATTRIBUTES_CONFIGURE

In the `tsZCL_CallBackEvent` structure (see Section 6.2) for this event:

- The `uMessage` field contains a structure of the type `tsZCL_AttributeReportingConfigurationRecord` (see Section 6.1.5).
- The `eZCL_Status` field indicates the outcome of parsing the configuration values for the attribute (success or failure)

Thus, the configuration of reporting for a set of attributes will result in a sequence of events of the above type, one for each attribute. The application should copy the contents of the `tsZCL_AttributeReportingConfigurationRecord` structure for each attribute to RAM (for information on storage format, refer to Appendix B.7.2).

Note that the `tsZCL_AttributeReportingConfigurationRecord` structure for an attribute contains the field `u16MaximumReportingInterval` which specifies a time-period for periodic reporting. Periodic reporting should not be too frequent, since a sleepy device must wake to send a report and frequent reports are be a significant drain on power resources. Therefore, the period for periodic reporting is not allowed to be set to a value less than `sConfig.u16SystemMaximumReportingInterval` in the ZCL configuration (see Appendix B.3.7). If a ‘configure reporting’ command attempts to set a smaller (non-zero) value, the ZCL discards the reporting configuration for this attribute and set the status for this attribute configuration to E_ZCL_CMDS_INVALID_VALUE in the ‘configure reporting’ response (see below).

Once attribute reporting has been configured (or not) for all the attributes (in the request), a single event is generated of the type:

E_ZCL_CBET_REPORT_ATTRIBUTES_CONFIGURE

Finally, the server generates a ‘configure reporting’ response and sends it back to the requesting client.

Note: *The application and ZCL hold the attribute reporting configuration data in RAM. To preserve this data through episodes of power loss, the application should also save the data to NVM, as described in Appendix B.7.*

3. Receiving a ‘Configure Reporting’ Response (on Client)

A ‘configure reporting’ response from the cluster server contains an Attribute Status Record for each attribute that was included in the corresponding ‘configure reporting’ command. For each attribute in the response, the ZCL on the client generates an event of the type:

E_ZCL_CBET_REPORT_INDIVIDUAL_ATTRIBUTES_CONFIGURE_RESPONSE

In the `tsZCL_CallBackEvent` structure (see Section 6.2) for this event, the `uMessage` field contains a structure of the type `tsZCL_AttributeReportingConfigurationResponse` (see Section 6.1.6). In this structure:

- The `eCommandStatus` field indicates the status of the attribute reporting configuration for the attribute.
- The `tsZCL_AttributeReportingConfigurationRecord` structure (Section 6.1.5) contains other data but only the following fields are used:
  - `u16AttributeEnum` which identifies the attribute
  - `u8DirectionIsReceived` which should read 0x01 to indicate that reports of the attribute value are received by the client

Once the above event has been generated for each valid attribute in the response, a single E_ZCL_CBET_REPORT_ATTRIBUTES_CONFIGURE_RESPONSE event is generated to conclude the response.

51.3.6 B.3.6: Configuring Default Reporting (on Server)

Default reporting is a form of automatic reporting for a restricted set of attributes (see Appendix B.2). It is configured on the cluster server.

An individual attribute can be configured as potentially reportable through default reporting by setting the 'reportable flag' E_ZCL_AF_RP in either of the following ways:

- The flag can be incorporated in the line for the attribute in the `tsZCL_AttributeDefinition` structure for the cluster server. For example, in the following line of code, attribute reporting is enabled for the `bOnOff` attribute of the On/Off cluster:

```
E_CLD_ONOFF_ATTR_ID_ONOFF, (E_ZCL_AF_RD|
E_ZCL_AF_SE|E_ZCL_AF_RP), E_ZCL_BOOL,
    (uint32)(&((tsCLD_OnOff*)(0))->bOnOff),0},
```

- The flag can be set by the server application by calling the function **eZCL_SetReportableFlag()**.

The reporting of these attributes can be configured by the server application by calling the function **eZCL_CreateLocalReport()** for each attribute. The configuration values are similar to those for automatic reporting, described in Appendix B.3.5. The reporting configuration is passed to the function in a `sZCL_AttributeReportingConfigurationRecord` structure. The application can then enable default reporting for reportable attributes using the function **vZCL_SetDefaultReporting()**, which checks whether the E_ZCL_AF_RP flag has been set for each attribute and, if so, sets the 'default reporting flag' E_ZCL_ACF_RP.

51.3.7 B.3.7: ZCL Configuration for Attribute Reporting

This section describes aspects of ZCL configuration related to attribute reporting.

Note: *The information in this section is only useful to developers who wish to adjust the standard ZCL configuration for attribute reporting.*

Each attribute for which automatic reporting is enabled requires a `tsZCL_ReportRecord` structure. These structures are maintained internally by the ZCL and space for them is allocated on the ZCL heap. The heap is allocated using the **u32ZCL_Heap** macro - for example:

PRIVATE uint32 u32ZCL_Heap[

ZCL_HEAP_SIZE(*ZCL_NUMBER_OF_ENDPOINTS***,**

*ZCL_NUMBER_OF_TIMERS***,**

*ZCL_NUMBER_OF_REPORTS***)];**

The number of reportable attributes and the maximum/minimum reporting intervals are passed into the internal `eZCL_CreateZCL` structure via the `sConfig` parameter - for example:

```
sConfig.u8NumberOfReports = ZCL_NUMBER_OF_REPORTS;
sConfig.u16SystemMinimumReportingInterval =
                            ZCL_SYSTEM_MIN_REPORT_INTERVAL;
sConfig.u16SystemMaximumReportingInterval =
                            ZCL_SYSTEM_MAX_REPORT_INTERVAL;
```

The above macros have default values that can be over-ridden in the application's **zcl_options.h** file, as indicated in Appendix B.3.1.

A server that supports automatic attribute reporting should have the 'reportable flag' E_ZCL_AF_RP set for any attributes that are reportable. While creating a cluster instance, the **vZCL_SetDefaultReporting()** function should be called, which will set the 'default reporting flag' E_ZCL_ACF_RP to enable default reporting for all the attributes that have the E_ZCL_AF_RP flag set. If a server receives a 'configure reporting' command for an attribute that does not have E_ZCL_ACF_RP flag set, it will return an error and not allow the attribute to be reported. This bit setting is also required for attribute reports generated through calls to the function **eZCL_ReportAllAttributes()**.

Attribute definitions will normally have the 'reportable flag' set only for the mandatory reportable attribute. The application on the server can set the E_ZCL_ACF_RP flag for those attributes on which reporting is not mandatory. This can be done using the function **eZCL_SetReportableFlag()**.

51.3.8 B.3.8: Speeding Up Automatic Attribute Reports

Automatic attribute reports (configured as described in Appendix B.3.5) that are produced on changes in attribute values can be speeded up to occur with millisecond resolution. Normally, these reports can occur on a timescale of seconds, as they are dependent on the E_ZCL_CBET_TIMER (one second) ticks for sampling. However, they can be made to occur on a timescale of milliseconds by providing E_ZCL_CBET_TIMER_MS (one millisecond) ticks.

In order to do this, the following code must be included in the application:

```
sCallBackEvent.eEventType = E_ZCL_CBET_TIMER_MS;
vZCL_EventHandler(&sCallBackEvent);
```

Note that the E_ZCL_CBET_TIMER ticks still need to be generated, as they are used by UTC time and by the ZCL report manager to keep track of time.

51.4 Appendix B.4: Sending attribute reports

If automatic attribute reporting has been configured between the cluster server and a client (as described in Appendix B.3), the reporting of the relevant attributes will begin immediately after configuration. Attribute reports are automatically generated:

- periodically with the configured time-interval between consecutive reports
- when the attribute value changes by at least the configured minimum amount

Automatic reporting normally employs both of the above mechanisms simultaneously but can be configured to operate without periodic reporting, if required.

If a periodic report becomes overdue, the event E_ZCL_CBET_REPORT_TIMEOUT is generated on the server.

The application on the server can also generate an attribute report, when needed, by calling one of the following functions:

- **eZCL_ReportAllAttributes()**, which sends an attribute report for all the reportable attributes
- **eZCL_ReportAttribute()**, which sends an attribute report for an individual reportable attribute

The above functions send an attribute report containing the current attribute value(s) to one or more clients specified in the function call. Only the standard attributes can be reported - this does not include manufacturer-specific attributes. Use of these functions for attribute reporting requires no special configuration on the server (but a recipient client will need attribute reporting to be enabled in its compile-time options).

Note: *The event E_ZCL_CBET_REPORT_REQUEST is automatically generated on the server before sending an attribute report, allowing the application to update the attribute values in the shared structure, if required.*

CAUTION: *The application must not rely on the above event as a prompt to update the shared structure when an attribute changes its value. The event is only generated when the change in attribute value is large enough for an attribute report to be produced. Smaller changes will not result in the event or a report.*

51.5 Appendix B.5: Receiving attribute reports

In order to receive and parse attribute reports from the cluster server, a client must have attribute reporting enabled in its compile-time options (see Appendix B.3.1).

When an attribute report is received from the server, events are generated and the ZCL software performs the following steps:

1. For each attribute in the attribute report, the ZCL generates an E_ZCL_CBET_REPORT_INDIVIDUAL_ ATTRIBUTE message for the endpoint callback function, which may or may not take action on this message.

2. On completion of the parsing of the attribute response, the ZCL generates a single E_ZCL_CBET_REPORT_ATTRIBUTES message for the endpoint callback function, which may or may not take action on this message.

Note that:

- The E_ZCL_CBET_REPORT_INDIVIDUAL_ATTRIBUTE event has the same fields as the E_ ZCL_CBET_READ_INDIVIDUAL_ATTRIBUTE_RESPONSE event. In the `uMessage` field of the `tsZCL_CallBackEvent` structure (see Section 6.2) for these events, the same structure is used, which is of the type `tsZCL_IndividualAttributesResponse`. However, the `eAttributeStatus` field is not updated for an attribute report (only for a 'read attributes' response).
- The E_ZCL_CBET_REPORT_ATTRIBUTES event has the same fields as the E_ZCL_CBET_READ_ATTRIBUTES_RESPONSE event.

51.6 Appendix B.6: Querying attribute reporting configuration

Any authorised device in a ZigBee wireless network can obtain the attribute reporting configuration of a cluster server. Such a query follows the process below:

1. The cluster client sends a 'read reporting configuration' command to the server.

2. The server receives and processes the command, retrieves the required configuration information and generates a 'read reporting configuration' response, which it sends back to requesting client.

3. The client receives the 'read reporting configuration' response and the ZCL generates events to inform the application of the reporting configuration.

These steps are described separately below.

Sending a 'Read Reporting Configuration' Command (from Client)

The application on the cluster client device can request the attribute reporting configuration on the server using **eZCL_SendConfigureReportingCommand()**. This function sends a 'read reporting configuration' command to the server.

In this function call, a `tsZCL_AttributeReadReportingConfigurationRecord` structure must be specified which indicates the required configuration information - this structure includes a pointer to an array of records, one per attribute for which reporting configuration information is needed (see Section 6.1.7).

Receiving a ‘Read Reporting Configuration’ Command (on Server)

The server automatically processes an incoming ‘read reporting configuration’ command without assistance from the application. Callback events are not generated. However, the server generates a ‘read reporting configuration’ response and send it back to the requesting client.

Receiving a ‘Read Reporting Configuration’ Response (on Client)

A ‘read reporting configuration’ response from the cluster server contains an Attribute Reporting Configuration Record for each attribute that was included in the corresponding ‘read reporting configuration’ command. For each attribute in the response, the ZCL on the client generates an event of the type:

E_ZCL_CBET_REPORT_READ_INDIVIDUAL_ATTRIBUTE_CONFIGURATION_RESPONSE

In the `tsZCL_CallBackEvent` structure (see Section 6.2) for this event, the `uMessage` field contains a structure of the type `tsZCL_AttributeReportingConfigurationResponse` (see Section 6.1.6) - this is the same structure as used in attribute reporting configuration, described in Appendix B.3.5.

In this structure:

- The `eCommandStatus` field indicates the status of the request.
- The `tsZCL_AttributeReportingConfigurationRecord` structure (see Section 6.1.5) includes:
  - `u16AttributeEnum` which identifies the attribute
  - other fields containing the attribute reporting configuration information

Once the above event has been generated for each valid attribute in the response, a single E_ZCL_CBET_REPORT_READ_ATTRIBUTE_CONFIGURATION_RESPONSE event is generated to conclude the response.

51.7 Appendix B.7: Storing an attribute reporting configuration

During the configuration of automatic attribute reporting, described in Appendix B.3.5, the application on the server must store attribute reporting configuration data in RAM and, optionally, in Non-Volatile Memory (NVM). The storage of this data is described in the sub-sections below.

51.7.1 Persisting an attribute reporting configuration

The attribute reporting configuration data is stored in RAM on the cluster server. To allow the server device to recover from an interruption of service involving a loss of power, this configuration data should also be saved in Non-Volatile Memory (NVM). In this case, the attribute reporting configuration data can be recovered from NVM during a ‘cold start’ of the device and automatic attribute reporting can resume without further configuration.

The storage of attribute reporting configuration data in NVM should be performed during the updates of this data on the server, described in Appendix B.3.5. When an E_ZCL_CBET_REPORT_INDIVIDUAL_ATTRIBUTES_CONFIGURE event is generated for an attribute, the contents of the incorporated structure `tsZCL_AttributeReportingConfigurationRecord` should be saved to NVM as well as to RAM (for information on storage format, refer to Appendix B.7.2). Data storage in NVM can be performed under application control using the Non-Volatile Memory Manager (NVM), described in the *Connectivity Framework Reference Manual*.

On a ‘cold start’ of the device, the application must retrieve the Attribute Reporting Configuration Record for each attribute from NVM and update the ZCL with the reporting configuration (this must be done after the ZCL has been initialized). To do this, the NVM can be used to retrieve the configuration record for an attribute and the function **eZCL_CreateLocalReport()** must then be called to register this data with the ZCL. This function must not be called for attributes that have not been configured for automatic attribute reporting (e.g. those for which the maximum reporting interval is set to REPORTING_MAXIMUM_TURNED_OFF).

Note: *The maximum reporting interval in NVM must be set to REPORTING_MAXIMUM_TURNED_OFF (0xFFFF) during a factory reset in order to prevent reporting from being enabled for attributes for which reporting was not previously enabled.*

51.7.2 Formatting an attribute reporting configuration record

The format in which the server application stores attribute reporting configuration data in RAM and, optionally, in NVM is at the discretion of the application developer.

The most general method is to store this data in an array of structures, in which there is one array element for each attribute for which automatic reporting is implemented (the size of this array should correspond to the value of the compile-time option SE_NUMBER_OF_REPORTS - see Appendix B.3.1). The information stored for each attribute may include the relevant cluster ID and endpoint number, as well as details of the configured change that can result in an attribute report. However, this method of data storage may require significant memory space and may only be necessary for more complex applications.

Alternative storage formats for this data are possible which economize on the memory requirements. These methods are outlined below.

Reduced Data Storage

A simple extension of the above general scheme uses application knowledge of the attributes being reported. In this case, certain static information about the reportable attributes is built into the compiled application and only the changeable information about these attributes is saved to an array in RAM (and NVM). In this way, the required memory space to store the attribute reporting configuration data is reduced.

An example of this method with five reportable attributes is given below.

```
#define SE_NUMBER_OF_REPORTS 5
typedef struct
  {
   uint16 u16Min;
   uint16 u16Max;
   tuZCL_AttributeReportable uChangeValue;
  } tsLocalStruct;
static tsLocalStruct asLocalConfigStruct[SE_NUMBER_OF_REPORTS];
typedef struct
 {
  uint16 u16AttEnum;
  teZCL_ZCLAttributeType eAttType;
 }tsLocalDefs;
static const tsLocalDefs asLocalDefs[SE_NUMBER_OF_REPORTS] = {
   {TPRC_MATCH_1,E_ZCL_UINT32},
   {TPRC_MATCH_6,E_ZCL_BMAP48},
   {TPRC_MATCH_7,E_ZCL_GINT56},
   {TPRC_MATCH_5,E_ZCL_UINT56},
   {TPRC_MATCH_3,E_ZCL_BOOL}
  };
```

In the above example:

- The fixed data (attribute identifier and type) is held in an array of `tsLocalDefs` structures, with one array element per attribute - this array is defined at compile-time and therefore does not need to be updated in RAM or persisted in NVM.
- The attribute reporting configuration data is held in an array of `tsLocalStruct` structures, with one array element per attribute - only this array needs to be updated in RAM and persisted in NVM, thus saving storage space.

Note that both arrays have SE_NUMBER_OF_REPORTS elements and there is a one-to-one correspondence between the elements of the two arrays - elements with the same number relate to the same attribute.

Minimized Data Storage

It may be possible to optimize the format in which the attribute reporting configuration data is saved in order to suit the attributes reported. For example, if there are only two attributes to be reported, then it may be sufficient to store the attribute reporting configuration data in a single structure, like the following:

```
typedef struct
{
    uint16   u16MinimumReportingIntervalForAttA;
    uint16   u16MaximumReportingIntervalForAttA;
    zint32   u32AttAReportableChange;
    uint16   u16MinimumReportingIntervalForAttB;
    uint16   u16MaximumReportingIntervalForAttB;
        // Attribute B is a discrete type (for example, a bitmap), so does not
 have a reportable change
} tsZCL_PersistedAttributeReportingConfigurationRecord;
```

52 Appendix C: Extended attribute discovery

'Extended' attribute discovery is similar to the normal attribute discovery described in Section 2.3.4 except the accessibility of each attribute is additionally indicated as being 'read', 'write' or 'reportable'. The application coding details and compile-time options are different, and are described below.

52.1 Appendix C.1: Compile-time options

If required, the extended attribute discovery feature must be explicitly enabled on the cluster server and client at compile-time by respectively including the following defines in the **zcl_options.h** files:

```
#define ZCL_ATTRIBUTE_DISCOVERY_EXTENDED_SERVER_SUPPORTED
#define ZCL_ATTRIBUTE_DISCOVERY_EXTENDED_CLIENT_SUPPORTED
```

52.2 Appendix C.2: Application coding

The application on a cluster client can initiate an extended attribute discovery on the cluster server by calling the **eZCL_SendDiscoverAttributesExtendedRequest()** function, which sends a 'discover attributes extended' request to the server. This function allows a range of attributes to be searched for, defined by:

- The 'start' attribute in the range (the attribute identifier must be specified).
- The number of attributes in the range.

Initially, the start attribute should be set to the first attribute of the cluster. If the discovery request does not return all the attributes used on the cluster server, the above function should be called again with the start attribute set to the next 'undiscovered' attribute. Multiple function calls may be required to discover all of the attributes used on the server.

On receiving a discover attributes extended request, the server handles the request automatically (provided that extended attribute discovery has been enabled in the compile-time options - see above) and replies with a 'discover attributes extended' response containing the requested information.

The arrival of the response at the client results in the event E_ZCL_CBET_DISCOVER_INDIVIDUAL_ATTRIBUTE_EXTENDED_RESPONSE for each attribute reported in the response. Therefore, multiple events normally result from a single discover attributes extended request. This event contains details of the reported attribute in a `tsZCL_AttributeDiscoveryExtendedResponse` structure (see Section 6.1.11).

Following the event for the final attribute reported, the event E_ZCL_CBET_DISCOVER_ATTRIBUTES_EXTENDED_RESPONSE is generated to indicate that all attributes from the discover attributes extended response have been reported.

53 Appendix D: Custom endpoints

A ZigBee device and its associated clusters can be registered on an endpoint using the relevant device registration function, from those listed and described in the *ZigBee Devices User Guide (JNUG3131)*. However, it is also possible to set up a custom endpoint which supports selected clusters (rather than a whole ZigBee 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 ZigBee device (such as a Light Sensor) while one or more custom endpoints are used to support selected clusters.

53.1 Appendix D.1: Devices and Endpoints

When using custom endpoints, it is important to note the difference between the following 'devices':

- **Physical device:** This is the physical entity which is the network node
- **Logical device:** This is a software entity which implements a specific set of functionality on the node, e.g. On/Off Switch device

A ZigBee network node may contain multiple endpoints, where one endpoint is used to represent the 'physical device' and other endpoints are used to support 'logical devices'. The following rules apply to cluster instances on endpoints:

- All cluster instances relating to a single 'logical device' must reside on a single endpoint.
- The Basic cluster relates to the 'physical device' rather than a 'logical device' instance. There can be only one Basic 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
  - A separate cluster instance on each 'logical device' endpoint, but each cluster instance must use the same `tsZCL_ClusterInstance` structure (and the same attribute values)

53.2 Appendix D.2: Cluster Creation Functions

For each of the following clusters, a creation function is provided which creates an instance of the cluster on an endpoint:

- Basic: **eCLD_BasicCreateBasic()**
- Power Configuration: **eCLD_PowerConfigurationCreatePowerConfiguration()**
- Device Temperature Configuration: **eCLD_DeviceTemperatureConfigurationCreateDeviceTemperatureConfiguration()**
- Identify: **eCLD_IdentifyCreateIdentify()**
- Groups: **eCLD_GroupsCreateGroups()**
- Scenes: **eCLD_ScenesCreateScenes()**
- On/Off: **eCLD_OnOffCreateOnOff()**
- On/Off Switch Configuration: **eCLD_OOSCCreateOnOffSwitchConfig()**
- Level Control: **eCLD_LevelControlCreateLevelControl()**
- Alarms: **eCLD_AlarmsCreateAlarms()**
- Time: **eCLD_TimeCreateTime()**
- Analogue Input (Basic): **eCLD_AnalogInputBasicCreateAnalogInputBasic()**
- Analogue Output (Basic): **eCLD_AnalogOutputBasicCreateAnalogOutputBasic()**
- Binary Input (Basic): **eCLD_BinaryInputBasicCreateBinaryInputBasic()**
- Binary Output (Basic): **eCLD_BinaryOutputBasicCreateBinaryOutputBasic()**
- Multistate Input (Basic): **eCLD_MultistateInputBasicCreateMultistateInputBasic()**
- Multistate Output (Basic): **eCLD_MultistateOutputBasicCreateMultistateOutputBasic()**
- Poll Control: **eCLD_PollControlCreatePollControl()**

- Power Profile: **eCLD_PPCreatePowerProfile()**
- Diagnostics: **eCLD_DiagnosticsCreateDiagnostics()**
- Illuminance Measurement: **eCLD_IlluminanceMeasurementCreateIlluminanceMeasurement()**
- Illuminance Level Sensing: **eCLD_IlluminanceLevelSensingCreateIlluminanceLevelSensing()**
- Temperature Measurement: **eCLD_TemperatureMeasurementCreateTemperatureMeasurement()**
- Pressure Measurement: **eCLD_PressureMeasurementCreatePressureMeasurement()**
- Flow Measurement: **eCLD_FlowMeasurementCreateFlowMeasurement()**
- Relative Humidity Measurement: **eCLD_RelativeHumidityMeasurementCreateRelativeHumidity Measurement()**
- Occupancy Sensing: **eCLD_OccupancySensingCreateOccupancySensing()**
- Electrical Measurement: **eCLD_ElectricalMeasurementCreateElectricalMeasurement()**
- Colour Control: **eCLD_ColourControlCreateColourControl()**
- Ballast Configuration: **eCLD_BallastConfigurationCreateBallastConfiguration()**
- Thermostat: **eCLD_ThermostatCreateThermostat()**
- Thermostat User Interface Configuration: **eCLD_ThermostatUIConfigCreateThermostatUIConfig()**
- Door Lock: **eCLD_DoorLockCreateDoorLock()**
- IAS Zone: **eCLD_IASZoneCreateIASZone()**
- IAS Ancillary Control Equipment (ACE): **eCLD_IASACECreateIASACE()**
- IAS Warning Device (WD): **eCLD_IASWDCreateIASWD()**
- Price: **eSE_PriceCreate()**
- Demand-Response and Load Control (DRLC): **eSE_DRLCCreate()**
- Simple Metering: **eSE_SMCreate()**
- Commissioning: **eCLD_CommissioningClusterCreateCommissioning()**
- Touchlink Commissioning: **eCLD_ZllCommissionCreateCommission()**
- Appliance Control: **eCLD_ApplianceControlCreateApplianceControl()**
- Appliance Identification: **eCLD_ApplianceIdentificationCreateApplianceIdentification()**
- Appliance Events and Alerts: **eCLD_ApplianceEventsAndAlertsCreateApplianceEventsAndAlerts()**
- Appliance Statistics: **eCLD_ApplianceStatisticsCreateApplianceStatistics()**
- Over-The-Air (OTA) Upgrade: **eOTA_Create()**

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 are described in the corresponding chapters of this manual.

53.3 Appendix D.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 Custom Endpoint Structure below.

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 D.2.

4. Call the ZCL function **eZCL_Register()** for the endpoint.

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 sIdentifyServer;
    tsZCL_ClusterInstance sOnOffCluster;
    tsZCL_ClusterInstance sDoorLockCluster;
} tsHA_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 `tsIdentify_CustomStruct` structure for the Identify cluster

Memory for tables or any other resources, if required by the cluster creation function

Note: *If a custom endpoint is to co-exist with a device endpoint, 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.*

54 Appendix E: Manufacturer-specific attributes and commands

This appendix describes how a manufacturer can add their own custom attributes and commands to a cluster. Attributes and commands are covered in separate sections.

54.1 Appendix E.1: Adding Manufacturer-specific Attributes

To add a manufacturer-specific attribute to a cluster:

1. Specify your manufacturer ID code in the Node descriptor. Do this in the ZPS Configuration Editor by clicking on **Node Descriptor** for the relevant node and editing the **Manufacturer Code** field on the **Properties** tab.

2. In the **zcl_options.h** file:

a) Define your manufacturer ID code - for example, for a code of 0x1234, add the line:

```
#define ZCL_MANUFACTURER_CODE 0x1234
```

b) Define the macro that enables the use of manufacturer-specific attributes for the cluster - this macro is cluster-specific but for the Electrical Measurement cluster, the relevant line is:

```
#define CLD_ELECTMEAS_ATTR_MAN_SPEC
```

c) Define an Attribute ID for the new attribute (you must not use a value already used by another attribute) - for example, to add an attribute with an ID of 0x0B00 to the Electrical Measurement cluster, the relevant line is:

```
#define E_CLD_ELECTMEAS_ATTR_ID_MAN_SPEC 0x0B00
```

3. Add the new attribute to the cluster structure in the cluster's header file - for example, the code below shows the attribute `i16ManufacturerSpecific` added to the Electrical Measurement cluster (in **ElectricalMeasurement.h**):

```
typedef struct
{
    zbmap32             u32MeasurementType;
#ifdef CLD_ELECTMEAS_ATTR_AC_FREQUENCY
    zuint16             u16ACFrequency;
#endif
#ifdef CLD_ELECTMEAS_ATTR_RMS_VOLTAGE
    zuint16             u16RMSVoltage;
#endif
#ifdef CLD_ELECTMEAS_ATTR_RMS_CURRENT
    zuint16             u16RMSCurrent;
#endif
#ifdef CLD_ELECTMEAS_ATTR_ACTIVE_POWER
    zint16              i16ActivePower;
#endif
#ifdef CLD_ELECTMEAS_ATTR_REACTIVE_POWER
    zint16             i16ReactivePower;
#endif
#ifdef CLD_ELECTMEAS_ATTR_APPARENT_POWER
    zuint16            u16ApparentPower;
#endif
#ifdef CLD_ELECTMEAS_ATTR_POWER_FACTOR
    zint8             i8PowerFactor;
#endif
#ifdef CLD_ELECTMEAS_ATTR_MAN_SPEC
```

```
    zint16              i16ManufacturerSpecific;
#endif
} tsCLD_ElectricalMeasurement;
```

4. Add the new attribute to the source (**.c**) file for the cluster, being careful to add the attribute in the correct sequential position - for example, the following code fragment shows the attribute `i16ManufacturerSpecific` added to the Electrical Measurement cluster (in **ElectricalMeasurement.c**):

```
const tsZCL_AttributeDefinition
asCLD_ElectricalMeasurementClusterAttributeDefinitions[] = {
/* ZigBee Cluster Library Version */
                :

#ifdef CLD_ELECTMEAS_ATTR_POWER_FACTOR
{E_CLD_ELECTMEAS_ATTR_ID_POWER_FACTOR,
E_ZCL_AF_RD,
E_ZCL_INT8,
(uint16)(&((tsCLD_ElectricalMeasurement*)(0))->i8PowerFactor),
0}, /* Optional */
#endif

/* Manufacturer-specific Read-only Attribute */
#ifdef CLD_ELECTMEAS_ATTR_MAN_SPEC
{E_CLD_ELECTMEAS_ATTR_ID_MAN_SPEC,
(E_ZCL_AF_RD|E_ZCL_AF_MS),
E_ZCL_INT16,
(uint16)(&((tsCLD_ElectricalMeasurement*)(0))->i16ManufacturerSpecific),
0}, /* Optional */
#endif

/* Manufacturer-specific Read/Write Attribute */
#ifdef CLD_ELECTMEAS_ATTR_MAN_SPEC
{E_CLD_ELECTMEAS_ATTR_ID_MAN_SPEC,
(E_ZCL_AF_RD|E_ZCL_AF_WR|E_ZCL_AF_MS),
E_ZCL_INT16,
(uint16)(&((tsCLD_ElectricalMeasurement*)(0))->i16ManufacturerSpecific),
0}, /* Optional */
#endif
```

Within your application code, you can remotely read the value of the new attribute using the following function call:

```
eStatus = eZCL_SendReadAttributesRequest(1, 1, <Add cluster here>, FALSE,
&sSendAddress, &u8SequenceNumber, u8NumAtts, TRUE, HA_MANUFACTURER_CODE,
<Add your attribute to the list here>);
```

54.2 Appendix E.2: Adding Manufacturer-specific Commands

To add a manufacturer-specific command to a cluster:

1. Ensure that a manufacturer ID code has been specified, as described in Appendix E.1.

2. In the **zcl_options.h** file, define a Command ID for the new command (you must not use a value already used by another command) - for example, to add a command with an ID of 0x20 to the Basic cluster, the relevant line is:

```
#define E_CLD_BASIC_CMD_MANU_SPEC 0x20
```

3. In your application code, introduce a handling routine for the new command into the command handler function that is registered when the cluster instance is created - for example, in the case of the Basic cluster, this handler function is **eCLD_BasicCommandHandler()**, which is registered when the function **eCLD_BasicCreateBasic()** is called. Do this as follows:

a) Define a handler routine specifically for the new command - for example, in the case of the Basic cluster, this function may be **eCLD_BasicHandleManuSpecCommand()** and has the prototype:

```
teZCL_Status eCLD_BasicHandleManuSpecCommand(
   ZPS_tsAfEvent            *pZPSevent,
   tsZCL_EndPointDefinition *psEndPointDefinition,
   tsZCL_ClusterInstance    *psClusterInstance);
```

b) Add the Command ID and the above command-specific handler function into the registered command handler function - for example, in the case of the Basic cluster, the code for **eCLD_BasicCommandHandler()** would be modified as shown in the fragment below:

```
PUBLIC teZCL_Status eCLD_BasicCommandHandler(
           ZPS_tsAfEvent       *pZPSevent,
           tsZCL_EndPointDefinition   *psEndPointDefinition,
           tsZCL_ClusterInstance    *psClusterInstance)
{
.
.
.
    // SERVER
    switch(u8CommandIdentifier)
    {
        case(E_CLD_BASIC_CMD_RESET_TO_FACTORY_DEFAULTS):
        {
eCLD_BasicHandleResetToFactoryDefaultsCommand(pZPSevent,    psEndPointDefinition, psClusterInstance);
            break;
        }
        case(E_CLD_BASIC_CMD_MANU_SPEC):
eCLD_BasicHandleManuSpecCommand(pZPSevent, psEndPointDefinition, psClusterInstance);
            break;
        default:
        {
            // unlock
            eZCL_ReleaseMutex(psEndPointDefinition);
            return(E_ZCL_FAIL);
            break;
        }
     }
       .
       .
       .
}
```

4. Add a command payload structure for the new command into the source (**.c**) file for the cluster - for example:

```
typedef struct
{
    uint8  u8PayloadField1;
    uint16 u16PayloadField2;
    uint16 u16PayloadField3;
    uint32 u32PayloadField4;
}tsMS_ManuSpecCommand;
```

5. Add a function for sending the command to a remote node into the header (**.h**) and source (**.c**) files for the cluster - for example, in the case of the Basic cluster, the function might be:

```
PUBLIC teZCL_Status eCLD_BasicCommandManuSpecSend(
                  uint8 u8SourceEndpoint,
                  uint8 u8DestinationEndpoint,
```

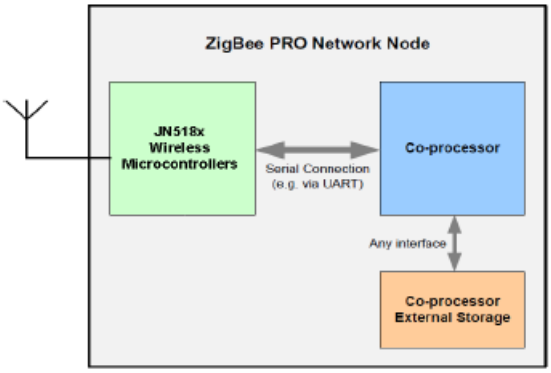
```
                    tsZCL_Address *psDestinationAddress,
                    tsMS_ManuSpecCommand *psManuSpecPayload,
              uint8 *pu8TransactionSequenceNumber)
{
        teZCL_Status eZCL_Status;
      tsZCL_TxPayloadItem asPayloadDefinition[] =
      {
        {1, E_ZCL_UINT8,  &psManuSpecPayload->u8PayloadField1},
        {1, E_ZCL_UINT16, &psManuSpecPayload->u16PayloadField2},
        {1, E_ZCL_UINT16, &psManuSpecPayload->u16PayloadField3},
        {1, E_ZCL_UINT32, &psManuSpecPayload->u32PayloadField4}
      };
      eZCL_Status = eZCL_CustomCommandSend(u8SourceEndpoint,
                  u8DestinationEndpoint,
                  psDestinationAddress,
                  GENERAL_CLUSTER_ID_BASIC,
                  TRUE,
                  E_CLD_BASIC_CMD_MANU_SPEC,
                  pu8TransactionSequenceNumber,
                  asPayloadDefinition,
                  TRUE,
                  HA_MANUFACTURER_CODE,
                  sizeof(asPayloadDefinition) / sizeof(tsZCL_TxPayloadItem));
    return eZCL_Status;
}
```

55 Appendix F: OTA extension for dual-processor nodes

This appendix describes use of the Over-the-Air (OTA) Upgrade cluster (introduced in Chapter 49) for a ZigBee PRO network consisting of dual-processor nodes that each contain a JN518x, K32W041, K32W061, MCXW71, or MCXW72 wireless microcontroller and a co-processor.

The co-processor is connected to the device via a serial interface and may have its own external storage device, as depicted in Figure 13 below.

Figure 12. Dual-Processor Node



The OTA Upgrade cluster may be used to upgrade the application which runs on the co-processor as well as the application which runs on the JN518x, K32W041, K32W061, MCXW71, or MCXW72 device. In this case, the OTA upgrade process is outlined below.

1. On the OTA server node (which is typically also the ZigBee Co-ordinator), the co-processor receives a new software image for the ZigBee PRO network.

2. The co-processor on the OTA server node saves the received software image in its own storage device.

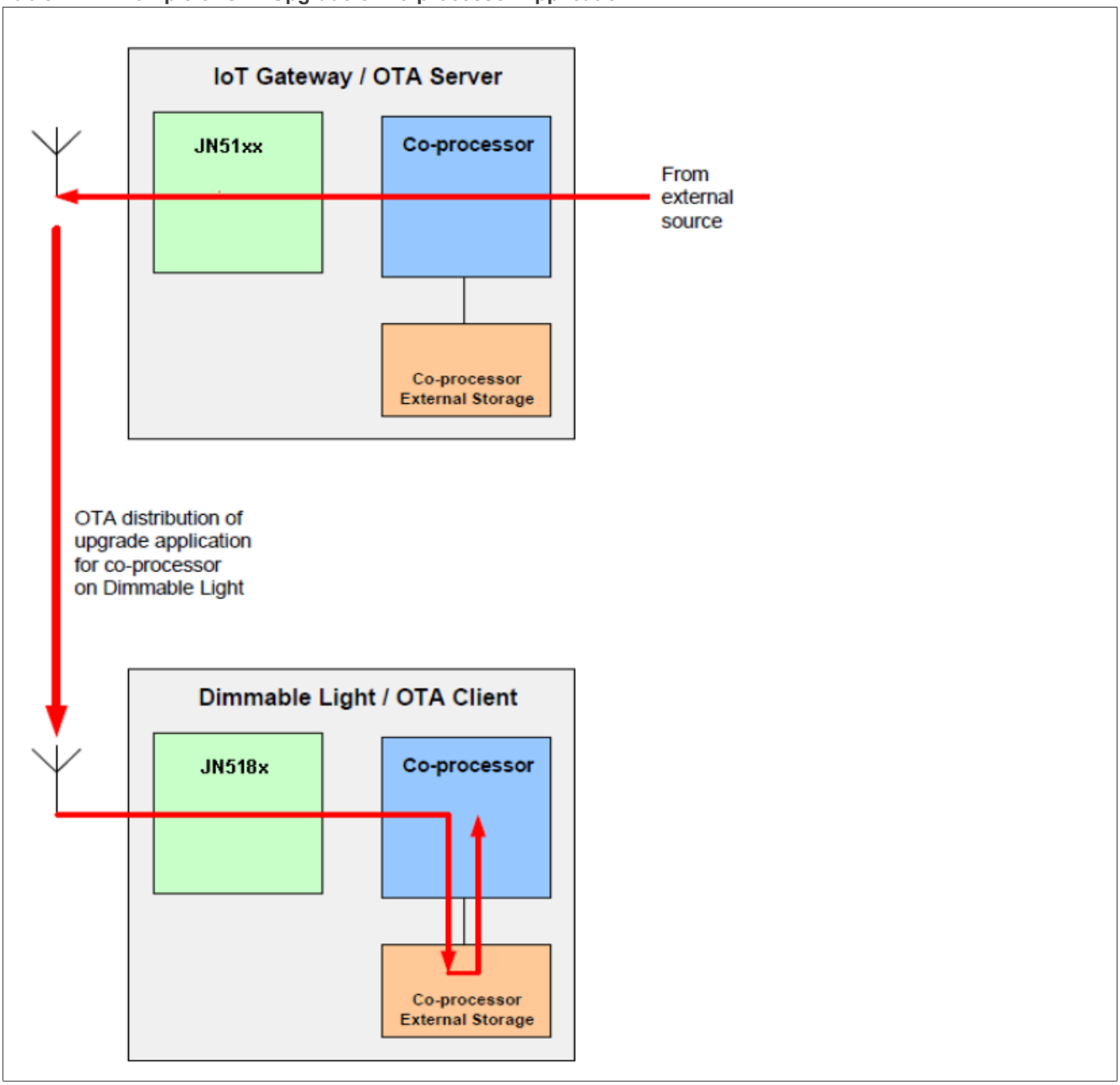
3. The OTA Upgrade cluster server running on the JN518x, K32W041, or K32W061 device distributes the software update over-the-air to the appropriate ZigBee PRO network nodes, as described in Section 49.4.

4. On a target node, the OTA Upgrade cluster client running on the JN518x/K32W041/61 microcontroller either stores the received software image in its own Flash memory device or passes it to the co-processor for storage in the co-processor's own storage device, depending on whether the application in the update is destined for the device or the co-processor.

5. The OTA Upgrade cluster client running on the JN518x, K32W041, or K32W061 device then either performs the upgrade of the application running on itself or signals to the co-processor to initiate an upgrade of its own application, as appropriate.

The above process is illustrated in Figure 14 below for the case of a ZigBee 3.0 network in which the co-processor application on a Dimmable Light (OTA client) is updated from an external source via an 'Internet of Things' (IoT) Gateway (OTA server) and the image is stored in the target co-processor's own storage device.

Table 124. Example of OTA Upgrade of Co-processor Application



55.1 Appendix F.1: Application Upgrades for Different Target Processors

In a ZigBee PRO network containing dual-processor nodes (with a JN518x, K32W041, or K32W061 Wireless microcontroller and a co-processor), an application upgrade can be targeted at any of the following processors:

- OTA server node processors:
  - Wireless Microcontroller (JN518x, K32W041, or K32W061)
  - Co-processor
- OTA client node processors:
  - Wireless Microcontroller (JN518x, K32W041, or K32W061)
  - Co-processor

Only application upgrades for the OTA client node processors need the new software image to be distributed over-the-air.

The following table describes the roles of the different processors (and their associated memory devices) during the different application upgrades.

Table 125. Processor Roles in Application Upgrade

Target Processor for Application Upgrade	Intermediate Processors during Application Upgrade			
	OTA Server		OTA Client	
	Co-processor	JN518x/K32 W041/K32W061	JN518x/K32 W041/K32W061	Co-processor
OTA Server Co-processor	Co-processor saves new image to its internal storage and performs update	-	-	-
OTA Server Wireless Micro-controller	Co-processor passes new image to server Wireless Microcontroller device *	Wireless Microcon-troller saves image to Flash memory and performs update *	-	-
OTA Client Wireless Micro-controller	Co-processor passes new image to server Wireless Microcontroller device *	Wireless Microcon-troller saves image to Flash memory and then sends it over-the-air to client *	Wireless Microcon-troller receives image, saves it to Flash memory and performs update	-
OTA Client Co-processor	Co-processor passes new image to server Wireless Microcontroller *	Wireless Microcon-troller saves image to Flash memory and then sends it over-the-air to client *	Wireless Microcon-troller receives image and saves it to Flash memory or to co-processor storage device	Co-processor per-forms update

* If insufficient space in Flash memory, image may be stored in co-processor storage - see Appendix F.2

The case of the co-processor on the OTA server node updating its own application is not described any further in this manual, as this upgrade mechanism is specific to the co-processor. The other three application upgrade scenarios are described in Appendix F.2.

55.2 Appendix F.2: Storing Upgrade Images in Co-processor Storage on Server

When the co-processor on the OTA server node receives a new OTA upgrade image from an external source (such as a utility company), if the image is not for the co-processor itself then it is normally passed to the Wireless Microcontroller device for storage in the attached Flash memory device. However, if there is insufficient storage space in Flash memory then the new image will need to be stored in the storage device of the co-processor:

- When the co-processor application notifies the Wireless Microcontroller application of the arrival of a new image, the Wireless Microcontroller application must check whether there is sufficient Flash memory space for the image.
- If there is insufficient Flash memory space, the Wireless Microcontroller application must inform the co-processor that it should store the image in its own storage device.

The maximum number of images that can be stored in the co-processor's storage device on the OTA server node must be specified as a compile-time option in the **zcl_options.h** file through the macro OTA_MAX_CO_PROCESSOR_IMAGES.

The OTA Upgrade cluster server will require knowledge of any OTA upgrade images stored in the co-processor's storage device - the cluster server must be able to advertise the availability of the image to cluster clients and be able to process requests for the image from clients. To facilitate this role, once the image has been saved, the co-processor must provide the OTA image header information to the Wireless Microcontroller application. The latter application can then register this header information with the cluster server by calling the function **eOTA_NewImageLoaded()**.

When an Image Block Request from a cluster client is received by the cluster server for an image stored in the co-processor's storage device, the event E_CLD_OTA_INTERNAL_COMMAND_CO_PRECOSSOR_IMAGE_BLOCK_REQUEST is generated on the Wireless Microcontroller. After requesting and receiving the required image block from the co-processor, the application must send the block to the relevant client by calling the function **eOTA_ServerImageBlockResponse()** to issue an Image Block Response.

55.3 Appendix F.3: Use of Image Indices

Each OTA upgrade image that is stored in non-volatile memory in a node is identified by an index number. This image index number is actually associated with the memory space allocated to a single image, rather than with a particular image. For example, the image index number `1` may correspond to sectors 3 and 4 of the Flash memory attached to the device.

Note: *In the case of external Flash memory, an image index number is linked with the start sector of the memory allocated to a single image when the function* ***eOTA_AllocateEndpointOTASpace()*** *is called.*

- The maximum number of images that can be stored in the external Flash memory is set at compile-time by defining a value for `OTA_MAX_IMAGES_PER_ENDPOINT` in the **zcl_options.h** file. The minimum value that can be used is 1, since the active image is held in the internal Flash memory and does not need to be included.
- Since the image indices are numbered from zero, they can take values in the range:
  – 0 to (`OTA_MAX_IMAGES_PER_ENDPOINT` - 1)

In the case of a dual-processor node, OTA upgrade images may also be stored in the co-processor's external storage device. The maximum number images that can be stored in this device is set at compile-time by defining a value for `OTA_MAX_CO_PROCESSOR_IMAGES` in the **zcl_options.h** file.

- The maximum number of images that can be stored across the two storage devices is:
  – `OTA_MAX_IMAGES_PER_ENDPOINT` + `OTA_MAX_CO_PROCESSOR_IMAGES`
- The image indices can take values in the range:
  – 0 to (`OTA_MAX_IMAGES_PER_ENDPOINT` + `OTA_MAX_CO_PROCESSOR_IMAGES` - 1)
- The indices of the images stored in the external Flash memory still take values in the range:
  – 0 to (`OTA_MAX_IMAGES_PER_ENDPOINT` - 1)
- The indices of the images stored in co-processor external storage take values in the range:
  – `OTA_MAX_IMAGES_PER_ENDPOINT` to (`OTA_MAX_IMAGES_PER_ENDPOINT` + `OTA_MAX_CO_PROCESSOR_IMAGES` - 1)

56 Appendix G: Glossary

Term	Description
Address	A numeric value that is used to identify a network node. In ZigBee, the device's 64-bit IEEE/MAC address or 16-bit network address is used.
AIB	APS Information Base: A database for the Application Support (APS) layer of the ZigBee stack, containing attributes concerned with system security.
APDU	Application Protocol Data Unit: Part of a wireless network message that is handled by the application and contains user data.
API	Application Programming Interface: A set of programming functions that can be incorporated in application code to provide an easy-to-use interface to underlying functionality and resources.
APS	Application Support: A sub-layer of the Application layer of the ZigBee stack, relating to communications with applications, binding and security.
Application	The program that deals with the input/output/processing requirements of the node, as well as high-level interfacing to the network.
Application Profile	A collection of device descriptors that characterise an application for a par-ticular market sector. An application profile can be public or private. A pub-lic profile is identified by a 16-bit number, assigned by the ZigBee Alliance.
Attribute	A data entity used by an application, e.g. a temperature measurement. It is part of a 'cluster' along with a set of commands which can be used to pass attribute values between applications or modify attributes.
Binding	The process of associating an endpoint on one node with an endpoint on another node, so that communications from the source endpoint are auto-matically routed to the destination endpoint without specifying addresses.
Channel	A narrow frequency range within the designated radio band - for example, the IEEE 802.15.4 2400-MHz band is divided into 16 channels. A wireless network operates in a single channel which is determined at network initial-isation.
Child	A node which is connected directly to a parent node and for which the par-ent node provides routing functionality. A child can be an End Device or Router. Also see Parent.
Cluster	A collection of attributes and commands that define a functional building block for a ZigBee device. The commands are used to communicate or modify attribute values. A cluster has input/ server and output/client sides - a cluster client issues a command which is received and acted on by a cluster server.
Context Data	Data which reflects the current state of the node. The context data must be preserved during sleep (of an End Device).
Co-ordinator	The node through which a network is started, initialised and formed - the Co-ordinator acts as the seed from which the network grows, as it is joined by other nodes. The Co-ordinator also usually provides a routing function. All networks must have one and only one Co-ordinator.
End Device	A node which has no networking role (such as routing) and is only con-cerned with data input/ output/processing. As such, an End Device cannot be a parent but can sleep to conserve power.
Endpoint	A software entity that acts as a communications port for an application on a ZigBee node. A node can support up to 240 endpoints, numbered 1 to 240. Two special endpoints are also supported - endpoint 0 is used by the ZDO and endpoint 255 is used for a broadcast to all endpoints on the node.
Extended PAN ID (EPID)	A 64-bit identifier for a ZigBee PRO network that is assigned when the net-work is started. A value can be pre-set or, alternatively, the IEEE/MAC address of the Co-ordinator can be used as the EPID.

Term	Description
IEEE 802.15.4	A standard network protocol that is used as the lowest level of the ZigBee software stack. Among other functionality, it provides the physical interface to the network's transmission medium (radio).
IEEE/MAC Address	A unique 64-bit address that is allocated to a device at the time of manufac-ture and is retained by the device for its lifetime. No two devices in the world can have the same IEEE/MAC address.
Joining	The process by which a device becomes a node of a network. The device transmits a joining request. If this is received and accepted by a parent node (Co-ordinator or Router), the device becomes a child of the parent. Note that the parent must have "permit joining" enabled.
Mesh Network	A wireless network topology in which all routing nodes (Routers and the Co-ordinator) can communicate directly with each other, provided that they are within radio range. This allows optimal and flexible routing, with alterna-tive routes if the most direct route is not available.
Network Address	A 16-bit address that is allocated to a ZigBee node when it joins a network. The Co-ordinator always has the network address 0x0000. In IEEE 802.15.4 terminology, it is called the short address.
NIB	NWK Information Base: A database containing attributes needed in the management of the Network (NWK) layer of the ZigBee stack.
Node Descriptor	A set of information about the capabilities of a node.
Node Power Descriptor	A set of information about a node's current and potential power supply.
NPDU	Network Protocol Data Unit: The transmitted form of a wireless network message (incorporates APDU and header/footer information from stack).
PAN ID	Personal Area Network Identifier: This is a 16-bit value that uniquely identi-fies the network - all neighbouring networks must have different PAN IDs.
Parent	A node which allows other nodes (children) to join the network through it and provides a routing function for these child nodes. A parent can be a Router or the Co-ordinator. Also see Child.
Router	A node which provides routing functionality (in addition to input/output/pro-cessing) if used as a parent node. Also see Routing.
Routing	The ability of a node to pass messages from one node to another, acting as a stepping stone from the source node to the target node. Routing function-ality is provided by Routers and the Co-ordinator. Routing is handled by the network level software and is transparent to the application on the node.
Simple Descriptor	A set of assorted information about a particular application/endpoint.
Sleep Mode	An operating state of a node in which the device consumes minimal power. During sleep, the only activity of the node may be to time the sleep duration to determine when to wake up and resume normal operation. Only End Devices can sleep.
Stack	The hierarchical set of software layers used to operate a system. The high-level user application is at the top of the stack and the low-level interface to the transmission medium is at the bottom of the stack.
Stack Profile	The set of features implemented from the ZigBee specification - that is, all the mandatory features together with a subset of the optional features. The ZigBee Alliance define two Stack Profiles for use with public Application Profiles - ZigBee and ZigBee PRO.
UART	Universal Asynchronous Receiver Transmitter: A standard interface used for cabled serial communications between two devices (each device must have a UART).
User Descriptor	A user-defined description of a node (e.g. "KitchenLight").
ZigBee Base Device	A framework for the use of ZigBee device types that provides basic func-tionality such as commissioning. Its functionality is defined in the ZigBee Base Device Behaviour (BDB) specification from the ZigBee Alliance.

Term	Description
ZigBee Certified Product	An end-product that uses ZigBee Compliant Platforms and public Applica-tion Profiles, and which has been tested for ZigBee compliance and subse-quently authorised to carry the ZigBee Alliance logo.
ZigBee Cluster Library (ZCL)	A collection of clusters that can be individually employed in ZigBee devices, as required, to implement the functionality of a device.
ZigBee Compliant Platform	A component (such as a module) that has been tested for ZigBee compli-ance and authorised to be used as a building block for a ZigBee end-prod-uct.
ZigBee Device Objects (ZDO)	A special application which resides in the Application Layer on all nodes and performs various standard tasks (e.g. device discovery, binding). The ZDO communicates via endpoint 0.

57 Revision history

The table below lists the revisions to this document.

Document ID	Release date	Description
3.1	24 January 2025	Added support for MCXW71 and MCXW72 devices
3.0	1 March 2023	• Added support for K32W1 device • Updated to latest NXP Documentation template
2.0	18 November 2019	Updated for JN518x and K32W041/K32W061
1.0	20 June 2018	First release

Legal information

Definitions

Draft — A draft status on a document indicates that the content is still under internal review and subject to formal approval, which may result in modifications or additions. NXP Semiconductors does not give any representations or warranties as to the accuracy or completeness of information included in a draft version of a document and shall have no liability for the consequences of use of such information.

Disclaimers

Limited warranty and liability — Information in this document is believed to be accurate and reliable. However, NXP Semiconductors does not give any representations or warranties, expressed or implied, as to the accuracy or completeness of such information and shall have no liability for the consequences of use of such information. NXP Semiconductors takes no responsibility for the content in this document if provided by an information source outside of NXP Semiconductors.

In no event shall NXP Semiconductors be liable for any indirect, incidental, punitive, special or consequential damages (including - without limitation - lost profits, lost savings, business interruption, costs related to the removal or replacement of any products or rework charges) whether or not such damages are based on tort (including negligence), warranty, breach of contract or any other legal theory.

Notwithstanding any damages that customer might incur for any reason whatsoever, NXP Semiconductors' aggregate and cumulative liability towards customer for the products described herein shall be limited in accordance with the Terms and conditions of commercial sale of NXP Semiconductors.

Right to make changes — NXP Semiconductors reserves the right to make changes to information published in this document, including without limitation specifications and product descriptions, at any time and without notice. This document supersedes and replaces all information supplied prior to the publication hereof.

Suitability for use — NXP Semiconductors products are not designed, authorized or warranted to be suitable for use in life support, life-critical or safety-critical systems or equipment, nor in applications where failure or malfunction of an NXP Semiconductors product can reasonably be expected to result in personal injury, death or severe property or environmental damage. NXP Semiconductors and its suppliers accept no liability for inclusion and/or use of NXP Semiconductors products in such equipment or applications and therefore such inclusion and/or use is at the customer's own risk.

Applications — Applications that are described herein for any of these products are for illustrative purposes only. NXP Semiconductors makes no representation or warranty that such applications will be suitable for the specified use without further testing or modification.

Customers are responsible for the design and operation of their applications and products using NXP Semiconductors products, and NXP Semiconductors accepts no liability for any assistance with applications or customer product design. It is customer's sole responsibility to determine whether the NXP Semiconductors product is suitable and fit for the customer's applications and products planned, as well as for the planned application and use of customer's third party customer(s). Customers should provide appropriate design and operating safeguards to minimize the risks associated with their applications and products.

NXP Semiconductors does not accept any liability related to any default, damage, costs or problem which is based on any weakness or default in the customer's applications or products, or the application or use by customer's third party customer(s). Customer is responsible for doing all necessary testing for the customer's applications and products using NXP Semiconductors products in order to avoid a default of the applications and the products or of the application or use by customer's third party customer(s). NXP does not accept any liability in this respect.

Terms and conditions of commercial sale — NXP Semiconductors products are sold subject to the general terms and conditions of commercial sale, as published at https://www.nxp.com/profile/terms, unless otherwise agreed in a valid written individual agreement. In case an individual agreement is concluded only the terms and conditions of the respective agreement shall apply. NXP Semiconductors hereby expressly objects to applying the customer's general terms and conditions with regard to the purchase of NXP Semiconductors products by customer.

Export control — This document as well as the item(s) described herein may be subject to export control regulations. Export might require a prior authorization from competent authorities.

Suitability for use in non-automotive qualified products — Unless this document expressly states that this specific NXP Semiconductors product is automotive qualified, the product is not suitable for automotive use. It is neither qualified nor tested in accordance with automotive testing or application requirements. NXP Semiconductors accepts no liability for inclusion and/or use of non-automotive qualified products in automotive equipment or applications.

In the event that customer uses the product for design-in and use in automotive applications to automotive specifications and standards, customer (a) shall use the product without NXP Semiconductors' warranty of the product for such automotive applications, use and specifications, and (b) whenever customer uses the product for automotive applications beyond NXP Semiconductors' specifications such use shall be solely at customer's own risk, and (c) customer fully indemnifies NXP Semiconductors for any liability, damages or failed product claims resulting from customer design and use of the product for automotive applications beyond NXP Semiconductors' standard warranty and NXP Semiconductors' product specifications.

Translations — A non-English (translated) version of a document, including the legal information in that document, is for reference only. The English version shall prevail in case of any discrepancy between the translated and English versions.

Security — Customer understands that all NXP products may be subject to unidentified vulnerabilities or may support established security standards or specifications with known limitations. Customer is responsible for the design and operation of its applications and products throughout their lifecycles to reduce the effect of these vulnerabilities on customer's applications and products. Customer's responsibility also extends to other open and/or proprietary technologies supported by NXP products for use in customer's applications. NXP accepts no liability for any vulnerability. Customer should regularly check security updates from NXP and follow up appropriately.

Customer shall select products with security features that best meet rules, regulations, and standards of the intended application and make the ultimate design decisions regarding its products and is solely responsible for compliance with all legal, regulatory, and security related requirements concerning its products, regardless of any information or support that may be provided by NXP.

NXP has a Product Security Incident Response Team (PSIRT) (reachable at PSIRT@nxp.com) that manages the investigation, reporting, and solution release to security vulnerabilities of NXP products.

Trademarks

Notice: All referenced brands, product names, service names, and trademarks are the property of their respective owners.

NXP — wordmark and logo are trademarks of NXP B.V.

Contents