ZigBee Cluster Library User Guide

JN-UG-3077

Revision 1.9

14 October 2013

Contents

Part II: Clusters and Modules

About this Manual

This manual describes the NXP implementation of the ZigBee Cluster Library (ZCL) for use with the Smart Energy (SE), Home Automation (HA) and ZigBee Light Link (ZLL) application profiles. The manual describes the clusters from the ZCL that may be used in SE, HA and ZLL applications.

Note: This manual assumes that you are already familiar with the concepts of ZigBee application profiles, devices, clusters and attributes. These are described in the *ZigBee PRO Stack User Guide (JN-UG-3048),* available from the NXP Wireless Connectivity TechZone (see “Support Resources” on page 19).

Organisation

This manual is divided into four parts:

- Part I: General and Development Information 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: Clusters and Modules comprises seventeen chapters (one chapter per cluster or module):
  - Chapter 5 details the Basic cluster
  - Chapter 6 details the Power Configuration cluster
  - Chapter 7 details the Identify cluster
  - Chapter 8 details the Groups cluster
  - Chapter 9 details the Scenes cluster
  - Chapter 10 details the On/Off cluster
  - Chapter 11 details the On/Off Switch Configuration cluster
  - Chapter 12 details the Level Control cluster
  - Chapter 13 details the Time cluster, as well as the use of ZCL time
  - Chapter 14 details the Binary Input (Basic) cluster
  - Chapter 15 details the Commissioning cluster
  - Chapter 16 details the Door Lock cluster

  - Chapter 17 details the Colour Control cluster
  - Chapter 18 details the Illuminance Measurement cluster
  - Chapter 19 details the Occupancy Sensing cluster
  - Chapter 20 details the Over-the-Air (OTA) Upgrade cluster
  - Chapter 21 details the EZ-mode Commissioning module
- Part III: General Reference Information comprises three chapters:
  - Chapter 22 details the general functions of the ZCL
  - Chapter 23 details the general structures used by the ZCL
  - Chapter 24 details the general enumerations used by the ZCL
- Part IV: Appendices describes the use of JenOS mutexes by the ZCL, the attribute reporting mechanism, the bootloaders for variants of the JN5148 wireless microcontroller and the OTA extension for dual-processor nodes, the terminology to use with EZ-mode commissioning, and also provides useful example code fragments.

Conventions

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

Function parameters are represented in *italics* type.

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

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

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

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

Acronyms and Abbreviations

API	Application Programming Interface
HA	Home Automation
OTA	Over the Air
SE	Smart Energy
ZCL	ZigBee Cluster Library
ZLL	ZigBee Light Link

Related Documents

JN-UG-3048	ZigBee PRO Stack User Guide
JN-UG-3059	ZigBee PRO Smart Energy API User Guide
JN-UG-3076	ZigBee Home Automation User Guide
JN-UG-3091	ZigBee Light Link User Guide
JN-UG-3075	JenOS User Guide
JN-UG-3081	Jennic Encryption Tool (JET) User Guide
075123	ZigBee Cluster Library Specification [from ZigBee Alliance]
095264	ZigBee Over-the-Air Upgrading Cluster [from ZigBee Alliance]

Support Resources

To access JN516x support resources such as SDKs, Application Notes and User Guides, visit the Wireless Connectivity TechZone:

www.nxp.com/techzones/wireless-connectivity

For JN514x resources, visit the NXP/Jennic web site: **www.jennic.com/support**

Trademarks

All trademarks are the property of their respective owners.

Chip Compatibility

The ZCL software described in this manual can be used on the following NXP wireless microcontrollers, depending on the ZigBee application profile used:

- **Smart Energy**
  - JN516x (currently only JN5168-001)
  - JN5148-Z01 (limited distribution)
- **Home Automation**
  - JN516x (currently only JN5168-001)
- **ZigBee Light Link**
  - JN516x (currently only JN5168-001)

Note: There are some implementation differences between the JN516x and JN5148 devices, particularly for the Over-The-Air (OTA) Upgrade cluster (described in Chapter 20).

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

Part I: General and Development Information

1. ZigBee Cluster Library (ZCL)

The ZigBee Alliance has defined the ZigBee Cluster Library (ZCL), comprising a number of standard clusters that can be applied to different functional areas. For example, all ZigBee application profiles use the Basic cluster from the ZCL.

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

The NXP implementation of the ZCL, described in this manual, is supplied in the NXP application profile software (e.g. JN-SW-4062 for ZigBee Light Link), available from the NXP Wireless Connectivity TechZone (see “Support Resources” on page 19). The ZCL is fully detailed in the *ZigBee Cluster Library Specification (075123)*, available from the ZigBee Alliance.

The supplied ZCL software can be used on the following NXP wireless microcontrollers:

- **Smart Energy**
  - JN516x (currently only JN5168-001)
  - JN5148-Z01 (limited distribution)
- **Home Automation**
  - JN516x (currently only JN5168-001)
- **ZigBee Light Link**
  - JN516x (currently only JN5168-001)

Note: There are some implementation differences between the JN516x and JN5148 devices, particularly for the Over-The-Air (OTA) Upgrade cluster (described in Chapter 20).

1.1 Member Clusters

The clusters of the ZCL include those listed in Table 1 below.

General Cluster	Cluster ID
Basic	0x0000
Power Configuration	0x0001
Identify	0x0003
Groups	0x0004
Scenes	0x0005
On/Off	0x0006
On/Off Switch Configuration	0x0007
Level Control	0x0008
Alarms	0x0009
Time	0x000A
Binary Input (Basic)	0x000F
Commissioning	0x0015
Door Lock	0x0101
Colour Control	0x0300
Illuminance Measurement	0x0400
Occupancy Sensing	0x0406

Table 1: ZCL Member Clusters

Note: Not all of the above clusters are currently available in the NXP implementation of the ZCL.

Basic

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

Power Configuration

The Power Configuration cluster allows the details of a device’s power source(s) to be determined and under/over voltage alarms to be configured. The Power Configuration cluster is detailed in Chapter 6.

Identify

The Identify cluster allows a ZigBee device to make itself known visually (e.g. by flashing a light) to an observer such as a network installer. The Identify cluster is detailed in Chapter 7.

Groups

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. The Groups cluster is detailed in Chapter 8.

Scenes

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. The Scenes cluster is detailed in Chapter 9.

On/Off

The On/Off cluster allows a device to be put into the ‘on’ and ‘off’ states, or toggled between the two states. The On/Off cluster is detailed in Chapter 10.

On/Off Switch Configuration

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. The On/Off Switch Configuration cluster is detailed in Chapter 11.

Level Control

The Level Control cluster allows control of the level of a physical quantity (e.g. heat output) on a device. The Level Control cluster is detailed in Chapter 12.

Alarms

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). The Alarms cluster is not yet supported in the NXP implementation of the ZCL.

Time

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 synchronise the clock to a time standard - the number of seconds since 0 hrs 0 mins 0 secs on 1st January 2000 UTC (Co-ordinated Universal Time). This cluster includes functionality for local time-zone and daylight saving time. The Time cluster is detailed in Chapter 13.

Binary Input (Basic)

The Binary Input (Basic) cluster provides an interface for accessing a binary measurement and its associated characteristics, and is typically used to implement a sensor that measures a two-state physical quantity. The Binary Input (Basic) cluster is detailed in Chapter 14.

Commissioning

The Commissioning cluster can be optionally used for commissioning the ZigBee stack on a device (during network installation) and defining the device behaviour with respect to the ZigBee network (it does not affect applications operating on the devices). The Commissioning cluster is detailed in Chapter 15.

Door Lock

The Door Lock cluster provides a means of representing the state of a door lock and (optionally) the door. The Door Lock cluster is detailed in Chapter 16.

Colour Control

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). The Colour Control cluster is detailed in Chapter 17.

Illuminance Measurement

The Illuminance Measurement cluster provides an interface to an illuminance measuring device, allowing the configuration of illuminance measuring and the reporting of illuminance measurements. The Illuminance Measurement cluster is detailed in Chapter 18.

Occupancy Sensing

The Occupancy Sensing cluster provides an interface to an occupany sensor, allowing the configuration of occupany sensing and the reporting of the occupancy status. The Occupancy Sensing cluster is detailed in Chapter 19.

Note: Some of the above clusters have special attributes that are used in ZigBee Light Link (ZLL) but in no other application profile. If required, these attributes must be enabled at compile-time (see Section 1.2).

1.2 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:

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

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 will apply to all clusters used in the application.

Tip: If only read access to attributes is required then do not enable write access, as omitting the write options will give the benefit of a reduced application size.

Optional and ZLL 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
```

The ZigBee Light Link (ZLL) application profile uses special attributes in the ZCL clusters. These attributes are not needed for other application profiles and must be enabled for ZLL by including the appropriate defines in the options header file.

Note: Cluster-specific compile-time options are detailed in the sections for the individual clusters in Chapter 5. The following optional features also have their own compile-time options: attribute reporting (see Appendix B.2.1) and OTA upgrade (see Section 20.12).

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. The attribute access functions are also detailed that are provided in the NXP implementation of the ZCL.

Note: ZCL functions are referred to in this chapter which are detailed in Chapter 22.

2.1 Shared Device Structures

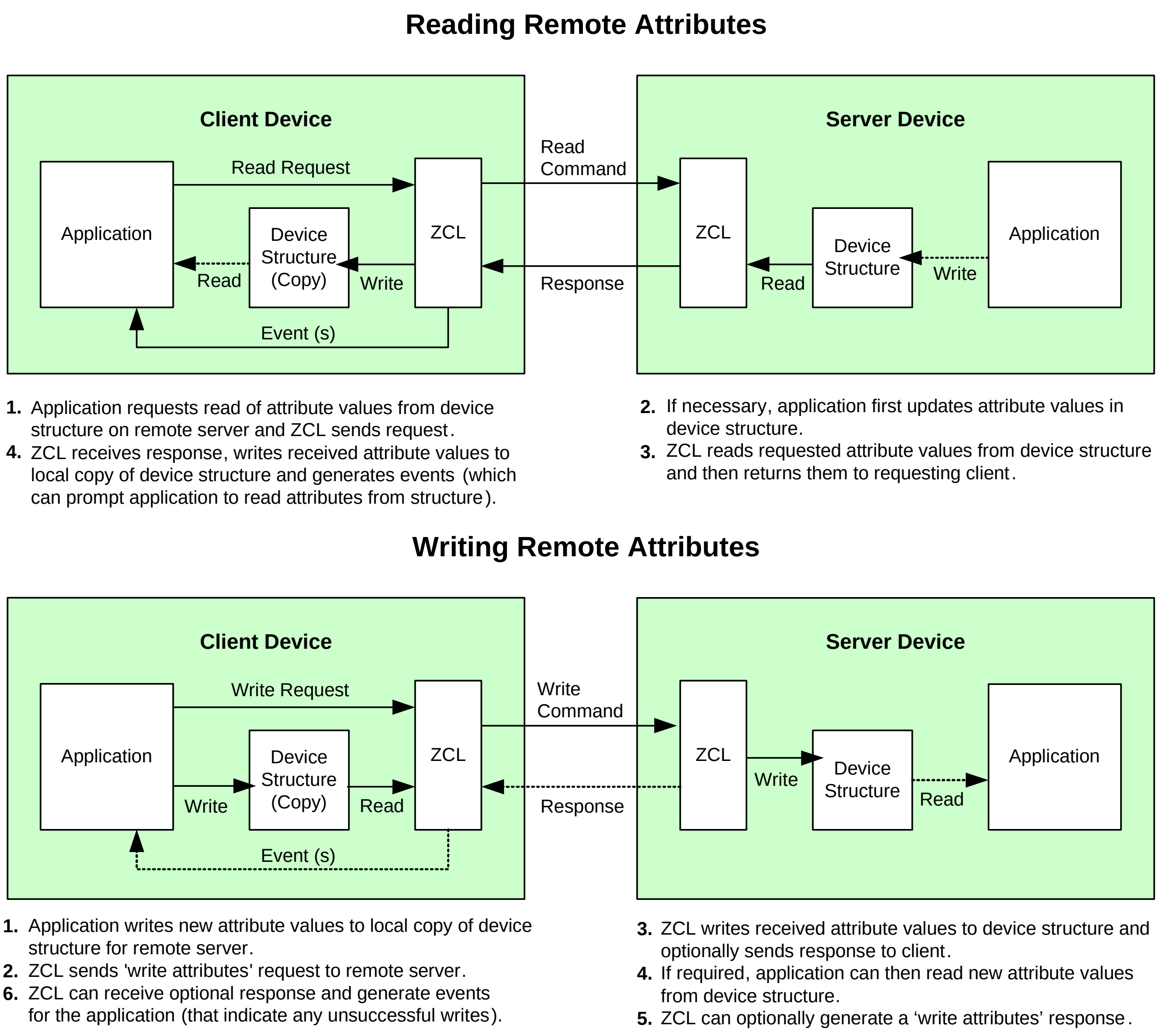
In each ZigBee device, cluster attribute values are exchanged between the application and the ZCL by means of a shared structure. This structure is protected by a mutex - see Appendix A. The structure for a particular ZigBee device contains structures for the clusters supported by that device.

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 1.2.

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

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

Remote read and write operations involving a shared device structure are illustrated in Figure 1 below. Normally, these operations are requested by a cluster client and performed on a cluster server. For more detailed descriptions of these operations, refer to Section 2.2.



1. Application requests read of attribute values from device structure on remote server and ZCL sends request.
4. ZCL receives response, writes received attribute values to local copy of device structure and generates events (which can prompt application to read attributes from structure).

2. If necessary, application first updates attribute values in device structure.
3. ZCL reads requested attribute values from device structure and then returns them to requesting client.

1. Application writes new attribute values to local copy of device structure for remote server.
2. ZCL sends 'write attributes' request to remote server.
6. ZCL can receive optional response and generate events for the application (that indicate any unsuccessful writes).

3. ZCL writes received attribute values to device structure and optionally sends response to client.
4. If required, application can then read new attribute values from device structure.
5. ZCL can optionally generate a 'write attributes' response.

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). The equivalent attributes of a cluster client on another device are copies of these cluster server attributes (remotely read from the server).

2.2 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 22.2.

2.2.1 Reading Attributes

A common operation in a ZigBee PRO application is to read attributes from a remote device, e.g. in a Smart Energy network, an In-Premise Display (IPD) device may need to obtain data from a Metering 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 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 that read access to cluster attributes must be explicitly enabled at compile-time as described in Section 1.2.

ZCL functions are provided for reading a set of attributes or all attributes of a remote cluster instance, as described in Section 2.2.1.1 and Section 2.2.1.2. A function is also provided for reading a local cluster attribute value, as described in Section 2.2.1.3.

2.2.1.1 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. Note that instances of the shared device structure (which contains the relevant attributes) exist on both the source and destination nodes. The events generated from a 'read attributes' request are further described in Chapter 3.

Note: The described sequence is similar when using the cluster-specific 'read attributes' functions and the **eZCL_ReadAllAttributes()** function.

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

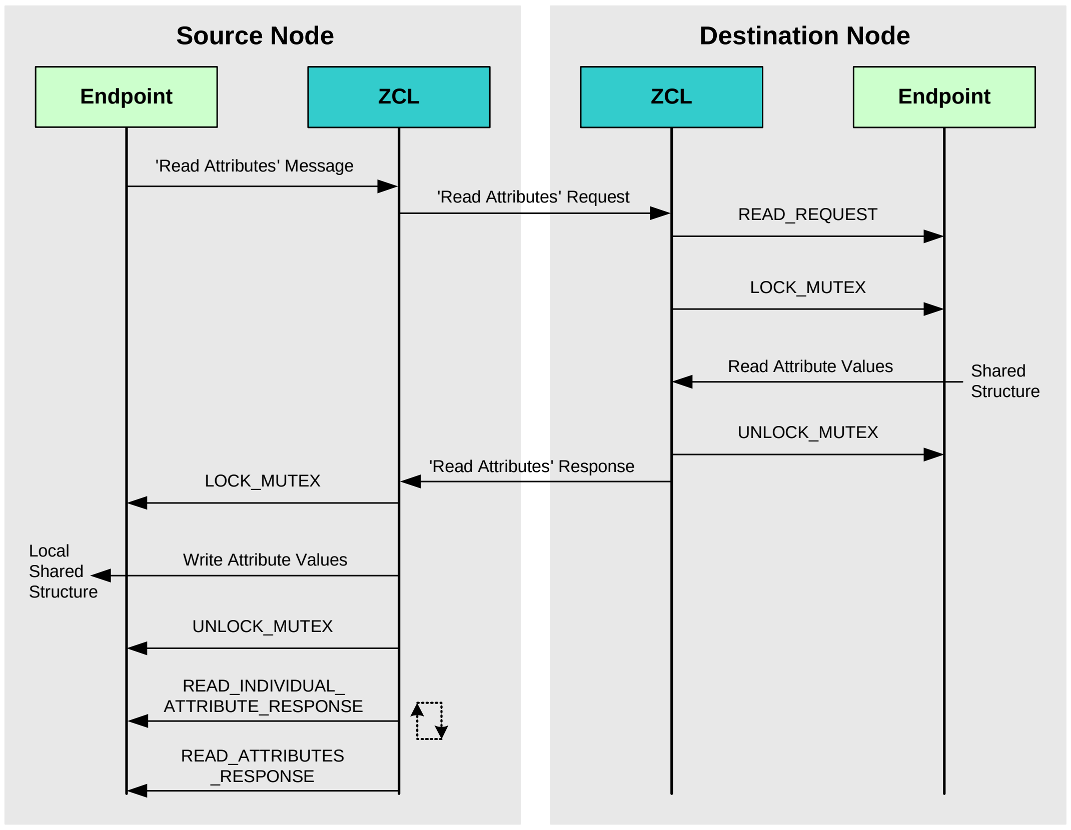


Figure 2: ‘Read Attributes’ Request and Response

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.

2.2.1.2 Reading All Attributes of a Remote Cluster

The function **eZCL_ReadAllAttributes()** allows a ‘read attributes’ request to be sent to a remote cluster in order to obtain the values of all server or client attributes, depending on the type of cluster instance (server or client).

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 that has been updated. Once all received attribute values have been parsed, an E_ZCL_CBET_READ_ATTRIBUTES_RESPONSE event is generated. The sequence is similar to that described in Section 2.2.1.1.

The response may not contain values for all requested attributes and so further responses may follow. The first E_ZCL_CBET_READ_ATTRIBUTES_RESPONSE should prompt the application to call **eZCL_HandleReadAttributesResponse()** in order to ensure that all cluster attributes are received from the remote node. This function should normally be included in the user-defined callback function that is invoked by the event E_ZCL_CBET_READ_ATTRIBUTES_RESPONSE. If the 'read attributes' response is not complete, this function will re-send 'read attributes' requests until all relevant attribute values have been received.

2.2.1.3 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).

2.2.2 Writing Attributes

The ZCL provides functions for writing attribute values to both remote and local clusters, as described in Section 2.2.2.1 and Section 2.2.2.2 respectively.

2.2.2.1 Writing to Attributes of a Remote Cluster

Some ZigBee PRO applications may need to write attribute values to a remote cluster - for example, in a Smart Energy network, an Energy Service Portal (ESP) may need to write attributes to a Load Control Device (e.g to configure the device group). 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. Note that write access to cluster attributes must be explicitly enabled at compile-time as described in Section 1.2.

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 3. Note that instances of the shared device structure (which contains the relevant attributes) must be maintained on both the source and destination nodes. The events generated from a ‘write attributes’ request are further described in Chapter 3.

1. On Source Node

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

1. Locks the mutex that protects the local instance of the shared device structure that contains the attributes to be updated - for information on mutexes, refer to Appendix A.
2. Writes one or more updated attribute values to the local instance of the shared device structure.
3. Unlocks the mutex that protects the local instance of the shared device structure.
4. Calls one of the above ZCL ‘write attributes’ functions to submit a request to update the relevant attributes on a cluster on a remote node. The information required by this function includes the following:
    - Source endpoint (from which the write request is to be sent)
    - Address of destination node for request
    - Destination endpoint (on destination node)
    - Identifier of the cluster containing the attributes [enumerations provided]
    - Number of attributes to be written
    - Array of identifiers of attributes to be written [enumerations provided]

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

2. On Destination Node

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

3. On Source Node

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.

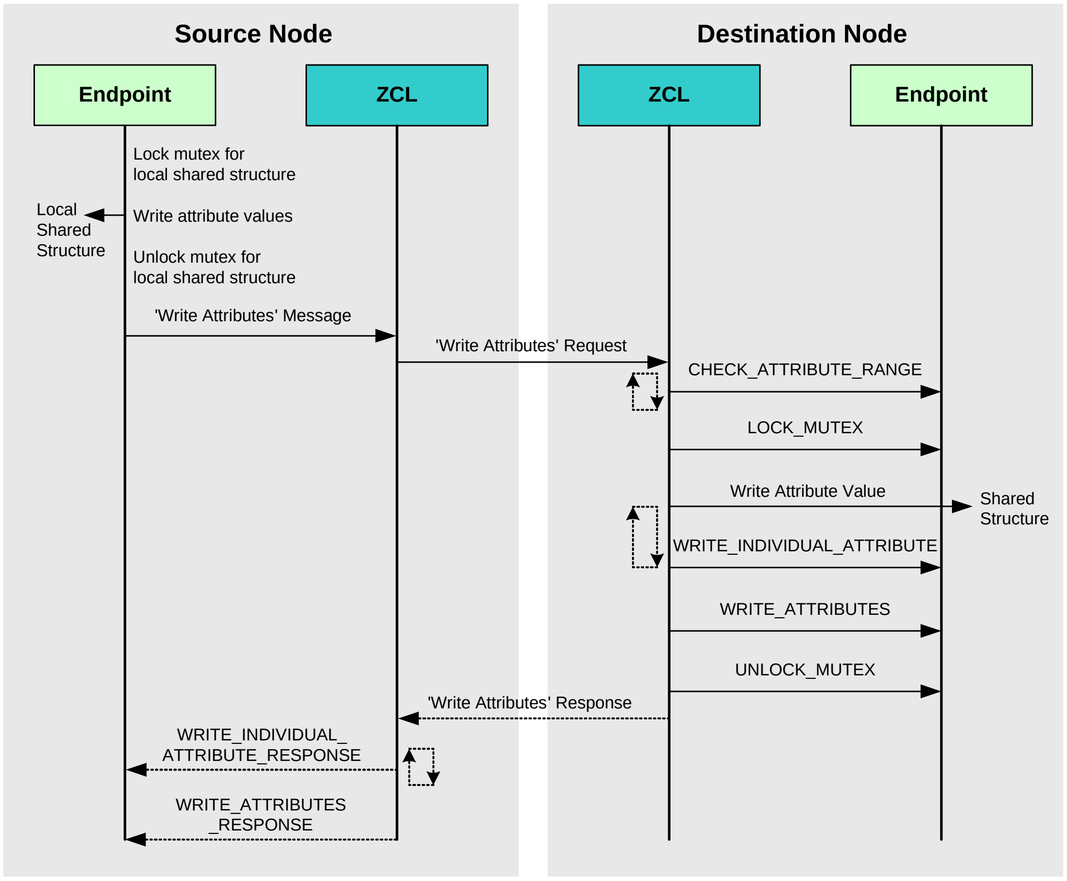

Source Node
Destination Node
Endpoint
ZCL
ZCL
Endpoint
Lock mutex for local shared structure
Local Shared Structure
Write attribute values
Unlock mutex for local shared structure
'Write Attributes' Message
'Write Attributes' Request
CHECK_ATTRIBUTE_RANGE
LOCK_MUTEX
Write Attribute Value
Shared Structure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

Figure 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.

2.2.2.2 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.

2.2.3 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.

For example, in the case of the Simple Metering cluster of the Smart Energy profile, those attributes corresponding to the quantities to be metered are enabled on the Metering Device which acts as the cluster server. An IPD, which is a cluster client, may only be able to display Current Summation and Instantaneous Demand. Instantaneous Demand is an optional attribute, so the IPD would need to discover whether the Metering Device supports it.

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

Note: 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 2.2.1) - if the attribute does not exist on the server, an error will be returned.

Compile-time Options

If required, the 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_SERVER_SUPPORTED
#define ZCL_ATTRIBUTE_DISCOVERY_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 will normally result from a single 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.

2.2.4 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 2.2.1. 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.

Note: This section only introduces attribute reporting. This optional feature is fully described in Appendix B.

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 is more fully described in Appendix B.1.

The rules for automatic reporting can be configured by a remote device by sending a 'configure reporting' command to the server using the function **eZCL_SendConfigureReportingCommand()**. If it is required, automatic attribute reporting must also be enabled at compile-time on both the cluster server and client. The configuration of attribute reporting is detailed in Appendix B.2.

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.6.

An attribute report for all attributes on the server can be issued directly by the server application using the function **eZCL_ReportAllAttributes()**. This method of attribute reporting does not require any configuration and does not need to be enabled at compile-time on the server, although the client still needs to be enabled at compile-time to receive attribute reports.

Sending an attribute report from the server is further described in Appendix B.3 and receiving an attribute report on the client is described in Appendix B.4.

2.3 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 23.1.1. 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 23.1.9). Therefore, in the case of an error, the identity of the command that caused the error will be contained in the command ID field of the default response.

Note that the default response can be generated on reception of all commands, including responses (e.g. a ‘read attributes’ response) but not other default responses.

2.4 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 PRO Stack User Guide (JN-UG-3048)*.

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 1: This feature for managing bound transmissions is not strictly a part of the ZCL but is provided in the ZCL software since it may be used with all ZigBee application profiles.

Note 2: 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 will be 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 will be 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
```

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 stack event is triggered by a message arriving in a message queue and a timer event is triggered when a JenOS 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 22.1. 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;
       tsZCL_AttributeStatusRecord          sReportingConfigurationResponse;
       tsZCL_ReportAttributeMirror          sReportAttributeMirror;
       uint32                               u32TimerPeriodMs;
   #ifdef EZ_MODE_COMMISSIONING
       tsZCL_EZModeBindDetails              sEZBindDetails;
   #endif
   }uMessage;
   ZPS_tsAfEvent                           *pZPSevent;
   tsZCL_ClusterInstance                   *psClusterInstance;
} tsZCL_CallBackEvent;
```

The fields of this structure are fully described Section 23.2.

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 will initially arrive as one of these events.

The occurrence of an event prompts JenOS to activate a ZCL user task - the event types and the task are pre-linked using the JenOS Configuration Editor. The following actions must then be performed in the application:

1. The task checks whether a message has arrived in the appropriate message queue, using the JenOS function **OS_eCollectMessage()**, or whether a JenOS timer has expired, using the JenOS function **OS_GetSWTimerStatus()**.
2. The task sets fields of the event structure `tsZCL_CallBackEvent` (see Section 3.1), as follows (all other fields are ignored):
   - If a timer event, sets the field `eEventType` to E_ZCL_CBET_TIMER
   - If a millisecond timer event, sets the field `eEventType` to E_ZCL_CBET_TIMER_MS
   - If 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 PRO Stack User Guide (JN-UG-3048)*
3. The task passes this event structure to the ZCL using **vZCL_EventHandler()** - the ZCL will then identify the event type (see Section 3.3) and invoke the appropriate endpoint callback function (for information on callback functions, refer to the documentation for the relevant application profile, e.g. Smart Energy).

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 will be invoked by the ZCL. If the event requires the attention of the application, the ZCL will replace the `eEventType` field with E_ZCL_CBET_CLUSTER_CUSTOM and populate the `tsZCL_ClusterCustomMessage` structure with the event data. The ZCL will then invoke 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.

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
	E_ZCL_CBET_ERROR
	E_ZCL_CBET_CLUSTER_UPDATE

Table 2: Events

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 JenOS 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 JenOS 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 2.2.1). 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.

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	

Table 3: Read Events

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 23.2.

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 2.2.2). 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.

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	

Table 4: Write Events

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 23.2.

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 will be allowed or disallowed

  The value to be written is pointed to by `pvAttributeData` in the above structure (note that 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 was successful, this status field is left as E_ZCL_SUCCESS.
  - If the write was unsuccessful, this status field will have been set to a suitable error status (see Section 24.1.4).

- **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 24.1.4).

- **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 events will be 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 will specify 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.

- **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 23.1.9) 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 22.1. The possible returned errors are listed in the Return/Status Codes chapter of the *ZigBee PRO Stack User Guide (JN-UG-3048)*.

4.2 Error/Command Status on Receiving Command

An error may be generated when a command is received by a device. If receiving a command results in an error, as indicated by an event of the type E_ZCL_CBET_ERROR on the device, 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 24.2.
- A 'default response' (see Section 23.1.9) may be generated which contains one of the command status codes detailed in Section 24.1.4. 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.

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_UNSUP_CLUSTER_COMMAND	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 is for all profiles but 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.
E_ZCL_ERR_KEY_ESTABLISHMENT_END_POINT_NOT_FOUND	None	Key Establishment cluster has not been registered correctly.
E_ZCL_ERR_KEY_ESTABLISHMENT_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.

Table 5: Error and Command Status Codes

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

Part II:
Clusters and Modules

5. Basic Cluster

This chapter details the Basic cluster which is defined in the ZCL and is a mandatory cluster for all ZigBee devices.

The Basic cluster has a Cluster ID of 0x0000.

5.1 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. NXP have implemented the Basic cluster in this way on the Smart Energy In-Premise Display (IPD) device.

The Basic cluster contains only two mandatory attributes, the remaining attributes being optional - see Section 5.2.

Note 1: The Basic cluster has an optional attribute which is only applicable to the ZigBee Light Link (ZLL) profile - see Section 5.2.

Note 2: 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 5.6.

5.2 Basic Cluster Structure and Attributes

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

```
typedef struct
{
    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_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

} tsCLD_Basic;
```

where:

- `u8ZCLVersion` is an 8-bit version number for the ZCL release that all clusters on the local endpoint(s) conform to. Currently, this should be set to 1
- `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 23.1.13) 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 23.1.13) 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 23.1.13) 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 profile/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 23.1.13 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 5.5.2

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 (an NXP plug-in for the Eclipse IDE).

- The following optional pair of attributes relates to the location of the device:
  - `sLocationDescription` is a `tsZCL_CharacterString` structure (see Section 23.1.13) 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
- `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 (this attribute may be used in the ZigBee Light Link profile only):
    - `sSWBuildID` is a `tsZCL_CharacterString` structure (see Section 23.1.13) 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

The Basic cluster structure contains two mandatory elements, `u8ZCLVersion` and `ePowerSource`. 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 5.3).

The mandatory attribute settings are described further in Section 5.3.

5.3 Mandatory Attribute Settings

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

On a mains-powered Smart Energy ESP/Meter:

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

On a battery-powered Smart Energy IPD:

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

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

5.4 Functions

The following Basic cluster function is provided in the NXP implementation of the ZCL:

Function	**Page**
eCLD_BasicCreateBasic	61

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 (e.g. IPD of the SE profile) 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 Basic cluster function called in the application, and must be called after the stack has been started and after the application profile has been initialised.

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

The array declaration should be as follows:

```
uint8
au8AppBasicClusterAttributeControlBits[CLD_BASIC_MAX_NUMBER_OF_ATTRIBUTE];
```

The function will initialise the array elements to zero.

Parameters

psClusterInstance	Pointer to structure containing information about the cluster instance to be created (see Section 23.1.15). This structure will be 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 23.1.2). 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 will initialise 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

5.5 Enumerations

5.5.1 teCLD_BAS_ClusterID

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

```
typedef enum PACK
{
    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;
```

5.5.2 teCLD_BAS_PowerSource

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

```
typedef enum PACK
{
     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.

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
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

Table 6: Power Source Enumerations

5.6 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.

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 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 (ZLL only)

```
#define   CLD_BAS_ATTR_SW_BUILD_ID
```

6. Power Configuration Cluster

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

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

6.1 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 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 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 6.5.

6.2 Power Configuration Cluster Structure and Attributes

The structure definition for the Power Configuration cluster is:

```
/* Power Configuration Cluster */
typedef struct
{

#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_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

} tsCLD_PowerConfiguration;
```

The attributes are classified into four attribute sets: Mains Information, Mains Settings, Battery Information and Battery Settings. 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)
2-7	Reserved

- `u16MainsVoltageMinThreshold` is the threshold for the under-voltage alarm, in units of 100 mV. The RMS mains voltage is allowed to dip 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

- `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.

Battery Settings Attribute Set

- `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 6.4.2.
- `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 threshold for the battery-low alarm, in units of 100 mV - the alarm is triggered when the battery voltage falls below this threshold.

6.3 Functions

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

Function	Page
eCLD_PowerConfigurationCreatePowerConfiguration	71

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 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 called after the stack has been started and after the application profile has been initialised.

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

The array declaration should be as follows:

```
uint8 au8AppPowerConfigurationClusterAttributeControlBits[
                                                  CLD_PWRCFG_MAX_NUMBER_OF_ATTRIBUTE];
```

The function will initialise the array elements to zero.

Parameters

psClusterInstance	Pointer to structure containing information about the cluster instance to be created (see Section 23.1.15). This structure will be 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 23.1.2). In this case, this structure must contain the details of the Basic cluster. This parameter can refer to a pre-filled structure called `sCLD_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 will initialise 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

6.4 Enumerations and Defines

6.4.1 teCLD_PWRCFG_AttributeId

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

```
typedef enum PACK
{
    E_CLD_PWRCFG_ATTR_ID_MAINS_VOLTAGE                = 0x0000,
    E_CLD_PWRCFG_ATTR_ID_MAINS_FREQUENCY,
    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,
    E_CLD_PWRCFG_ATTR_ID_BATTERY_VOLTAGE              = 0x0020,
    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,
} teCLD_PWRCFG_AttributeId;
```

6.4.2 teCLD_PWRCFG_BatterySize

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

```
typedef enum PACK
{
    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;
```

6.4.3 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)
```

6.5 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.

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 attribute:

```
#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
```

7. Identify Cluster

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

The Identify cluster has a Cluster ID of 0x0003.

7.1 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 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 Identify cluster are fully detailed in Section 7.9.

Note: The Identify cluster contains optional functionality for the EZ-mode Commissioning module, which is detailed in Chapter 21 (and is currently only available for use with the Home Automation profile). *However, this enhanced functionality is not presently certifiable.*

7.2 Identify Cluster Structure and Attribute

The structure definition for the Identify cluster is:

```
typedef struct
{
    zuint16      u16IdentifyTime;

#ifdef CLD_IDENTIFY_ATTR_COMMISSION_STATE
    zbmap8       u8CommissionState;
#endif
} tsCLD_Identify;
```

where:

- `u16IdentifyTime` is a mandatory attribute specifying the remaining length of time, in seconds, that the device will continue in identification mode. Setting the attribute to a non-zero value will put the device into identification mode and the attribute will subsequently be decremented every second
- `u8CommissionState` is an optional attribute for use with EZ-mode Commissioning (see Chapter 21) to indicate the network status and operational status of the node - this information is contained in a bitmap, as follows:

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 will also be set to 1) • 0 if not commissioned for operation
2 - 7	Reserved

7.3 Initialisation

The function **eCLD_IdentifyCreateIdentify()** is used to create an instance of the Identify cluster. This function is generally called by the initialisation 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 (amongst others).

7.4 Sending Commands

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

7.4.1 Starting and Stopping Identification Mode

The function **eCLD_IdentifyCommandIdentifyRequestSend()** can be 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 7.7.2):

- 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 currently in progress on the server device.

In a ZigBee Light Link (ZLL) network, identification mode can alternatively be started and stopped as described in Section 7.4.2.

7.4.2 Requesting Identification Effects (ZLL Only)

The function **eCLD_IdentifyCommandTriggerEffectSend()** can be used in a ZigBee Light Link (ZLL) network to request a particular identification effect or behaviour on a light of a remote node (this function can be used for entering and leaving identification mode instead of **eCLD_IdentifyCommandIdentifyRequestSend()**).

The possible behaviours 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 one-second period, and then this is repeated 15 times
- **Okay:**
  - Colour light goes green for one second
  - Monochrome light flashes twice in one 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 (e.g. for the Breathe effect, only the current one-second cycle will be completed)
- **Stop effect:** Current effect and identification mode are terminated as soon as possible

7.4.3 Inquiring about Identification Mode

The function **eCLD_IdentifyCommandIdentifyQueryRequestSend()** can be called on an Identify cluster client in order to request a response from a server cluster if it is currently in identification mode. This request should only be unicast.

7.4.4 Using EZ-mode Commissioning Features (HA only)

When using the EZ-mode Commissioning module, which is described in Chapter 21 (and is currently only available with the Home Automation profile), the Identify cluster is mandatory:

- An EZ-mode initiator device must host an Identify cluster client
- An EZ-mode target device must host an Identify cluster server

The Identify cluster also contains the following optional features that can be used with the EZ-mode Commissioning module (*these features are not currently certifiable*).

‘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 the requested commissioning action(s). The local application must perform these action(s) using the functions of the EZ-mode Commissioning module (see Section 21.5). 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 7.9).

‘Commissioning State’ Attribute

The Identify cluster server contains an optional ‘Commissioning State’ attribute, `u8CommissionState` (see Section 7.2), 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 7.9).

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.

7.5 Sleeping Devices in Identification Mode

If a device sleeps between activities (e.g. a switch that is configured as a sleeping End Device) and is also operating in identification mode, the device must wake once per second for the ZCL to decrement the *u16IdentifyTime* attribute (see Section 7.2), which represents the time remaining in identification mode. The device may also use this wake time to highlight itself, e.g. flash a LED. The attribute update is performed automatically by the ZCL when the application passes an E_ZCL_CBET_TIMER event to the ZCL via the **vZCL_EventHandler()** function. The ZCL will also automatically increment ZCL time as a result of this event.

When in identification mode, it is not permissible for a device to sleep for longer than one second and to generate one timer event on waking. Before entering sleep, the value of the *u16IdentifyTime* attribute can be checked - if this is zero, the device is not in identification mode and is therefore allowed to sleep for longer than one second (for details of updating ZCL time following a prolonged sleep, refer to Section 13.4.1).

7.6 Functions

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

Function	Page
eCLD_IdentifyCreateIdentify	82
eCLD_IdentifyCommandIdentifyRequestSend	84
eCLD_IdentifyCommandTriggerEffectSend	86
eCLD_IdentifyCommandIdentifyQueryRequestSend	88
eCLD_IdentifyEZModeInvokeCommandSend	90
eCLD_IdentifyUpdateCommissionStateCommandSend	92

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 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 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 must not be called for an endpoint on which a standard ZigBee 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 Identify cluster function called in the application, and must be called after the stack has been started and after the application profile has been initialised.

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

The array declaration should be as follows:

```
uint8
au8AppIdentifyClusterAttributeControlBits[CLD_IDENTIFY_MAX_NUMBER_OF_ATTRIBU
TE];
```

The function will initialise the array elements to zero.

Parameters

psClusterInstance	Pointer to structure containing information about the cluster instance to be created (see Section 23.1.15). This structure will be 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 23.1.2). 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 will initialise 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 structure which contains custom data for the Identify cluster (see Section 7.7.1). This structure is used for internal data storage. No knowledge of the fields of this structure is required

Returns

E_ZCL_SUCCESS

E_ZCL_ERR_PARAMETER_NULL

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 7.7.2). The required duration of the identification mode is specified in the payload and this value will replace the value in the Identify cluster structure on the target device.

A device which receives this command will generate 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 will be set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

Parameters

u8SourceEndPointId	Number of the local endpoint through which 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 will be 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 will be 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 7.7.2).

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()**.

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 a ZigBee Light Link (ZLL) network, 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()**.

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 one-second period, and then this is repeated 15 times
Okay	• Colour light goes green for one second • Monochrome light flashes twice in one 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 (e.g. for the Breathe effect, only the current one-second cycle will be 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 will generate 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 will be set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

Parameters

u8SourceEndPointId	Number of the local endpoint through which 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 will be 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 will be 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 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()**.

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 will generate a callback event on the endpoint on which the Identify cluster was registered. If the receiving device is currently in identification mode, it will return a response containing the amount of time for which it will continue in this mode (see Section 7.7.3).

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 will be set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

Parameters

u8SourceEndPointId	Number of the local endpoint through which 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 will be 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 will be 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()**.

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 (for more information, refer to Chapter 21):

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 7.7.4). 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 will be generated with the required commissioning action(s) specified in the `u8Action` field of the `tsCLD_Identify_EZModeInvokePayload` structure. It is the local application's responsibility to perform the requested action(s) using the functions of the EZ-mode Commissioning module (see Section 21.5).

Note that the 'EZ-mode Invoke' command is optional and, if required, must be enabled in the compile-time options (see Section 7.9).

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 will be set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

Parameters

u8SourceEndPointId	Number of the local endpoint through which 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 will be 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 will be 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 7.7.4)

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()**.

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 7.2) which is used for EZ-mode commissioning. The command allows individual bits of `u8CommissionState` to be set or cleared (see Section 7.7.4).

On receiving the 'Update Commission State' command on the target device, an event will be generated and the requested update will be automatically performed.

Note that the `u8CommissionState` attribute is optional and, if required, must be enabled in the compile-time options (see Section 7.9).

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 will be set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

Parameters

u8SourceEndPointId	Number of the local endpoint through which 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 will be 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 will be 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 7.7.4)

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()**.

7.7 Structures

7.7.1 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.

7.7.2 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 currently in identification mode, the mode will be terminated immediately.

7.7.3 Custom Command Responses

The following structure contains the response to a query as to whether a device is currently 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 will remain in identification mode.

7.7.4 EZ-mode Commissioning Command Payloads

The structures shown and described below may be used when the Identify cluster is used in conjunction with the EZ-mode Commissioning module.

‘EZ-Mode Invoke’ Command Payload

The following structure is used when sending an ‘EZ-mode Invoke’ command (using the **eCLD_IdentifyEZModeInvokeCommandSend()** function).

```
typedef struct
{
    zbmap8    u8Action;
} tsCLD_Identify_EZModeInvokePayload;
```

where `u8Action` is a bitmap specifying the EZ-mode commissioning action(s) 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 7.2).

```
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 bit(s) to ‘1’
  - 2: Clear the specified bit(s) 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

7.8 Enumerations

7.8.1 teCLD_Identify_ClusterID

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

```
typedef enum PACK
{
    E_CLD_IDENTIFY_ATTR_ID_IDENTIFY_TIME = 0x0000,   /* Mandatory */
    E_CLD_IDENTIFY_ATTR_ID_COMMISSION_STATE          /* Optional */
 } teCLD_Identify_ClusterID;
```

7.9 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 optional cluster functionality can be enabled in the **zcl_options.h** file.

Enhanced Functionality for EZ-mode Commissioning (HA only)

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
```

Note that the above EZ-mode Commissioning features are not currently certifiable.

Enhanced Functionality for ZLL

Enhanced functionality (identification effects) is available for the ZigBee Light Link (ZLL) profile - see Section 7.4.2. To enable this enhanced cluster functionality for ZLL, you must include:

```
#define CLD_IDENTIFY_SUPPORT_ZLL_ENHANCED_COMMANDS
```

8. Groups Cluster

This chapter describes the Groups cluster which is defined in the ZCL and allows the management of the Group table concerned with group addressing.

The Groups cluster has a Cluster ID of 0x0004.

8.1 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 delivered 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 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 Groups cluster are fully detailed in Section 8.8.

8.2 Groups Cluster Structure and Attribute

The structure definition for the Groups cluster is:

```
typedef struct
{
    zbmap8                  u8NameSupport;
} 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

8.3 Initialisation

The function **eCLD_GroupsCreateGroups()** is used to create an instance of the Groups cluster. The function is generally called by the initialisation 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 already exist, the function will create it. Therefore, this is a way of creating a local group.

8.4 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.

8.4.1 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 endpoint(s) to be added to the group, where the required group is specified in the payload of the command. If the group does not already exist in the target device's Group table, it will be added to the table.

- **eCLD_GroupsCommandAddGroupRequestSend()** can be used to request the addition of the target endpoint(s) to the specified group.
- **eCLD_GroupsCommandAddGroupIfIdentifyingRequestSend()** can be used to request the addition of the target endpoint(s) to the specified group provided that the target device is currently in identification mode of the Identity cluster (see Chapter 7).

An endpoint can also be added to a local group, as described in Section 8.3.

8.4.2 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 endpoint(s) to be removed from the group(s). If a group is empty following the removal of the endpoint(s), it will be deleted in the Group table.

- **eCLD_GroupsCommandRemoveGroupRequestSend()** can be used to request the removal of the target endpoint(s) 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(s) 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 will also result in the removal of the endpoint from the scene.

8.4.3 Obtaining Information about Groups

Two functions are provided for obtaining information about groups. Each function sends a command to the endpoint(s) 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.

8.5 Functions

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

Function	Page
eCLD_GroupsCreateGroups	102
eCLD_GroupsAdd	104
eCLD_GroupsCommandAddGroupRequestSend	105
eCLD_GroupsCommandViewGroupRequestSend	107
eCLD_GroupsCommandGetGroupMembershipRequestSend	109
eCLD_GroupsCommandRemoveGroupRequestSend	111
eCLD_GroupsCommandRemoveAllGroupsRequestSend	113
eCLD_GroupsCommandAddGroupIfIdentifyingRequestSend	115

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 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 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: This function must not be called for an endpoint on which a standard ZigBee 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 Groups cluster function called in the application, and must be called after the stack has been started and after the application profile has been initialised.

The function retrieves any group IDs already stored in the ZigBee PRO stack's Application Information Base (AIB). However, the AIB does not store group names. If name support is required, the application should store the group names using the JenOS PDM 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 23.1.15). This structure will be 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 23.1.2). 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 will initialise the attributes with default values.
psCustomDataStructure	Pointer to a structure containing the storage for internal functions of the cluster (see Section 8.6.1)
psEndPointDefinition	Pointer to the ZCL endpoint definition structure for the application (see Section 23.1.1)

Returns

E_ZCL_SUCCESS

E_ZCL_ERR_PARAMETER_NULL

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 will be 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 8.8).

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()**.

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 endpoint(s) 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 number(s).

The device receiving this message will generate a callback event on the endpoint on which the Groups cluster was registered and, if possible, add the group to its Group table before sending a response indicating success or failure (see Section 8.6.4).

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 will be set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

Parameters

u8SourceEndPointId	Number of the local endpoint through which 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 will be 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 will be 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 8.6.3)

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()**.

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 will generate a callback event on the endpoint on which the Groups cluster was registered and will generate a View Group response containing the group name (see Section 8.6.4).

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 will be set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

Parameters

u8SourceEndPointId	Number of the local endpoint through which 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 will be 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 will be 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 8.6.3)

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()**.

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 will generate a callback event on the endpoint on which the Groups cluster was registered and will generate a Get Group Membership response containing the required information (see Section 8.6.4).

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 will be set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

Parameters

u8SourceEndPointId	Number of the local endpoint through which 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 will be 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 will be 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 8.6.3)

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()**.

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 endpoint(s) from a particular group - that is, remove the endpoint(s) from the group's entry 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 will generate a callback event on the endpoint on which the Groups cluster was registered. If the group becomes empty following the deletion(s), the device will remove the group ID and group name from its Group table. It will then generate an appropriate Remove Group response indicating success or failure (see Section 8.6.4).

If the target endpoint belongs to a scene associated with the group to be removed (requiring the Scenes cluster - see Chapter 9), the endpoint will also be removed from this scene as a result of this function call - that is, the relevant scene entry will be 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 will be set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

Parameters

u8SourceEndPointId	Number of the local endpoint through which 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 will be 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 will be 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 8.6.3)

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()**.

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 endpoint(s) - that is, remove the endpoint(s) from all group entries in the Group table on the device and, if no other endpoints remain in a group, remove the group from the table.

The device receiving this message will generate a callback event on the endpoint on which the Groups cluster was registered. If a group becomes empty following the deletion(s), the device will remove the group ID and group name from its Group table.

If the target endpoint belongs to scenes associated with the groups to be removed (requiring the Scenes cluster - see Chapter 9), the endpoint will also be removed from these scenes as a result of this function call - that is, the relevant scene entries will be 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 will be set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

Parameters

u8SourceEndPointId	Number of the local endpoint through which 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 will be 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 will be 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()**.

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 endpoint(s) on the target device be added to a particular group on the condition that the remote device is currently 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 number(s). The identifying functionality is controlled using the Identify cluster (see Chapter 7).

The device receiving this message will generate a callback event on the endpoint on which the Groups cluster was registered and will then check whether the device is currently identifying itself. If so, the device will (if possible) add the group ID and group name to its Group table. If the device it not currently identifying itself then no action will be 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 will be set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

Parameters

u8SourceEndPointId	Number of the local endpoint through which 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 will be 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 will be 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 8.6.3)

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()**.

8.6 Structures

8.6.1 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 8.6.2.

8.6.2 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.

8.6.3 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 endpoint(s) must be added
- `sGroupName` is the name of the group to which the endpoint(s) 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 endpoint(s) must be removed

8.6.4 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 endpoint(s) 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 device's Group table to receive more groups - that is, the number of groups that may be added (special values: 0xFE means at least one more group may be added, a higher value means that the table's remaining capacity 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 endpoint(s) were removed

8.7 Enumerations

8.7.1 teCLD_Groups_ClusterID

The following structure contains the enumeration used to identify the attribute of the Groups cluster.

```
typedef enum PACK
{
    E_CLD_GROUPS_ATTR_ID_NAME_SUPPORT        = 0x0000   /* Mandatory */
} teCLD_Groups_ClusterID;
```

8.8 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.

Add this line to set the size used for the group addressing table in the **.zpscfg** file:

```
#define CLD_GROUPS_MAX_NUMBER_OF_GROUPS             (8)
```

Add this line to configure the maximum length of the group name:

```
#define CLD_GROUPS_MAX_GROUP_NAME_LENGTH            (16)
```

9. Scenes Cluster

This chapter describes the Scenes cluster which is defined in the ZCL.

The Scenes cluster has a Cluster ID of 0x0005.

9.1 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 (e.g. Night or Day mode for a lighting network in a Home Automation system). 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 8. 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 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 Scenes cluster are fully detailed in Section 9.9.

9.2 Scenes Cluster Structure and Attributes

The structure definition for the Scenes cluster is:

```
typedef struct
{
    zuint8                  u8SceneCount;
    zuint8                  u8CurrentScene;
    zuint16                 u16CurrentGroup;
    zbool                   bSceneValid;
    zuint8                  u8NameSupport;

#ifdef CLD_SCENES_ATTR_LAST_CONFIGURED_BY
    zieeeaddress            u64LastConfiguredBy
#endif

} 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)

9.3 Initialisation

The function **eCLD_ScenesCreateScenes()** is used to create an instance of the Scenes cluster. The function is generally called by the initialisation function for the host device.

9.4 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: In the case of the ZigBee Light Link profile, commands can also be issued for operations on the local node, as described in Section 9.5.

9.4.1 Creating a Scene

In order to create a scene, an entry for the scene must be added to the Scene table on every device that contains a cluster which is associated with the scene.

The function **eCLD_ScenesCommandAddSceneRequestSend()** can be used to request that a scene is added to a Scene table on a remote device. A call to this function can send 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.

In the case of the ZigBee Light Link profile, the enhanced function **eCLD_ScenesCommandEnhancedAddSceneRequestSend()** must be used instead, which allows the transition time for the scene to be set in units of tenths of a second (rather than seconds).

Alternatively, a scene can be created by saving the current attribute settings of the relevant clusters - in this way, the current state of the system (e.g. lighting levels in a Home Automation system) can be captured as a scene and re-applied ‘at the touch of a button’ when required. The current settings are stored as a scene in the Scene table using the function **eCLD_ScenesCommandStoreSceneRequestSend()** which, again, can send the request to a single device or multiple devices. If a Scene table entry already 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()**.

9.4.2 Copying a Scene (ZLL Only)

In the case of the ZigBee Light Link profile, scene settings can be copied from one scene to another scene on the same remote endpoint using the function **eCLD_ScenesCommandCopySceneSceneRequestSend()**. 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 already exists in the Scene table on the endpoint, the entry settings will be overwritten with the copied settings. Otherwise, a new Scene table entry will be 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 will be associated with two different group IDs.

9.4.3 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 will remain unchanged in the implementation of the cluster - that is, the corresponding parts of the system will not change their states.

9.4.4 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 will remove all scenes not associated with a group.

9.4.5 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.

  In the case of the ZigBee Light Link profile, the enhanced function **eCLD_ScenesCommandEnhancedViewSceneRequestSend()** must be used instead, 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.

9.5 Issuing Local Commands

Some of the operations described in Section 9.4 that correspond to remote commands can also be performed locally, as described below.

9.5.1 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 already exists in the table, the existing entry will be 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 already exists in the table, the existing entry will be overwritten with the exception of the transition time and scene name fields.

9.5.2 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 will remain unchanged.

9.6 Functions

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

Function	**Page**
eCLD_ScenesCreateScenes	129
eCLD_ScenesAdd	131
eCLD_ScenesStore	132
eCLD_ScenesRecall	133
eCLD_ScenesCommandAddSceneRequestSend	134
eCLD_ScenesCommandViewSceneRequestSend	136
eCLD_ScenesCommandRemoveSceneRequestSend	138
eCLD_ScenesCommandRemoveAllScenesRequestSend	140
eCLD_ScenesCommandStoreSceneRequestSend	142
eCLD_ScenesCommandRecallSceneRequestSend	144
eCLD_ScenesCommandGetSceneMembershipRequestSend	146
eCLD_ScenesCommandEnhancedAddSceneRequestSend	148
eCLD_ScenesCommandEnhancedViewSceneRequestSend	150
eCLD_ScenesCommandCopySceneSceneRequestSend	152

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 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 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: This function must not be called for an endpoint on which a standard ZigBee 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 Scenes cluster function called in the application, and must be called after the stack has been started and after the application profile has been initialised.

On calling this function for the first time, a ‘global scene’ entry is created/reserved in the Scene table. On subsequent calls (e.g. 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 will be no reinitialisation of the scene data. Note that removing all groups from the device will also remove 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 Scenes cluster, which can be obtained by using the macro CLD_SCENES_MAX_NUMBER_OF_ATTRIBUTE.

The array declaration should be as follows:

```
uint8
au8AppScenesClusterAttributeControlBits[CLD_SCENES_MAX_NUMBER_OF_ATTRIBUTE];
```

The function will initialise the array elements to zero.

Parameters

psClusterInstance	Pointer to structure containing information about the cluster instance to be created (see Section 23.1.15). This structure will be 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 23.1.2). 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 will initialise 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 9.7.1)
psEndPointDefinition	Pointer to the ZCL endpoint definition structure for the application (see Section 23.1.1)

Returns

E_ZCL_SUCCESS

E_ZCL_ERR_PARAMETER_NULL

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 already exists in the table, the existing entry will be overwritten (i.e. all previous scene data in this entry will be 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

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 already exists in the table, the existing entry will be overwritten (i.e. previous scene data in this entry will be lost), with the exception of the transition time field and the scene name field - these fields will be 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

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 will remain 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

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 9.7.2). The scene may also be associated with a particular group.

The device receiving this message will generate a callback event on the endpoint on which the Scenes cluster was registered and, if possible, add the scene to its Scene table before sending an Add Scene response indicating success or failure (see Section 9.7.3).

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 will be set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

Parameters

u8SourceEndPointId	Number of the local endpoint through which 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 will be 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 will be 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 9.7.2)

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()**.

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 will generate a callback event on the endpoint on which the Scenes cluster was registered and will generate a View Scene response containing the relevant information (see Section 9.7.3).

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 will be set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

Parameters

u8SourceEndPointId	Number of the local endpoint through which 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 will be 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 will be 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 9.7.2)

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()**.

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 will generate a callback event on the endpoint on which the Scenes cluster was registered. The device will then delete the scene in the Scene table. If the request was sent to a single device (rather than to a group address), it will then generate an appropriate Remove Scene response indicating success or failure (see Section 9.7.3).

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 will be set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

Parameters

u8SourceEndPointId	Number of the local endpoint through which 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 will be 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 will be 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 9.7.2)

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()**.

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 will remove all scenes not associated with a group.

The device receiving this message will generate a callback event on the endpoint on which the Scenes cluster was registered. The device will then delete the scenes in the Scene table. If the request was sent to a single device (rather than to a group address), it will then generate an appropriate Remove All Scenes response indicating success or failure (see Section 9.7.3).

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 will be set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

Parameters

u8SourceEndPointId	Number of the local endpoint through which 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 will be 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 will be 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 9.7.2)

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()**.

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 will be stored using the scene ID and group ID specified in the payload of the command. If an entry already exists with these IDs, its existing cluster settings will be overwritten with the new settings.

Note that the transition time and scene name fields are not set by this command (or for a new entry, they are set to null values). When using this command to create a new scene which requires particular settings for these fields, the scene entry must be created in advance using the Add Group command, at which stage these fields should be pre-configured.

The device receiving this message will generate a callback event on the endpoint on which the Scenes cluster was registered. If the request was sent to a single device (rather than to a group address), it will then generate an appropriate Store Scene response indicating success or failure (see Section 9.7.3).

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 will be set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

Parameters

u8SourceEndPointId	Number of the local endpoint through which 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 will be 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 will be 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 9.7.2)

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()**.

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 will remain unchanged in the implementation of the cluster.

The device receiving this message will generate a callback event on the endpoint on which the Scenes cluster was registered. If the request was sent to a single device (rather than to a group address), it will then generate an appropriate Recall Scene response indicating success or failure (see Section 9.7.3).

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 will be set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

Parameters

u8SourceEndPointId	Number of the local endpoint through which 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 will be 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 will be 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 9.7.2)

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()**.

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 will generate a callback event on the endpoint on which the Scenes cluster was registered. If the request was sent to a single device (rather than to a group address), it will then generate an appropriate Get Scene Membership response indicating success or failure and, if successful, the response will contain a list of the scene IDs associated with the given group ID (see Section 9.7.3). If the original command is sent to a group address, an individual device will only respond if it has scenes associated with the group ID in the command payload (so will only respond in the case of success).

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 will be set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

Parameters

u8SourceEndPointId	Number of the local endpoint through which 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 will be 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 will be 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 9.7.2)

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()**.

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 ZLL 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 can be used only with the ZLL profile and 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 9.7.2). The scene may also be associated with a particular group.

The device receiving this message will generate a callback event on the endpoint on which the Scenes cluster was registered and, if possible, add the scene to its Scene table before sending an Enhanced Add Scene response indicating success or failure (see Section 9.7.3).

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 will be set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

Parameters

u8SourceEndPointId	Number of the local endpoint through which 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 will be 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 will be 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 9.7.2)

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()**.

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 ZLL device, requesting information on a particular scene on the destination endpoint. The function can be used only with the ZLL profile and the returned information includes the finer transition time available with ZLL. 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 will generate a callback event on the endpoint on which the Scenes cluster was registered and will generate a Enhanced View Scene response containing the relevant information (see Section 9.7.3).

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 will be set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

Parameters

u8SourceEndPointId	Number of the local endpoint through which 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 will be 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 will be 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 9.7.2)

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()**.

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 ZLL 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 function can be used only with the ZLL profile. The relevant source and destination scene ID/group ID combinations are specified in the command payload.

Note that:

- If the destinaton scene ID/group ID already exists on the target endpoint, the existing scene will be overwritten with the new settings.
- The message payload contains a ‘copy all scenes’ bit which, if set to ‘1’, 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 will generate a callback event on the endpoint on which the Scenes cluster was registered and, if the original request was unicast, will generate a Copy Scene response (see Section 9.7.3).

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 will be set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

Parameters

u8SourceEndPointId	Number of the local endpoint through which 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 will be 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 will be 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 9.7.2)

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()**.

9.7 Structures

9.7.1 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.

9.7.2 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 will take 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 (ZLL Only)

```
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 ZLL device will take 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 (ZLL Only)

```
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 (ZLL Only)

```
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 will be 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 will be 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)

9.7.3 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 will take 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 device's Scene table to receive more scenes - that is, the number of scenes that may be added (special values: 0xFE means at least one more scene may be added, a higher value means that the table's remaining capacity 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 (ZLL Only)

```
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 (ZLL Only)

```
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 will take 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 (ZLL Only)

```
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` was the source group ID for the copy
- `u8FromSceneId` was the source scene ID for the copy

9.8 Enumerations

9.8.1 teCLD_Scenes_ClusterID

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

```
typedef enum PACK
{
    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;
```

9.9 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.

Add this line to enable the optional Last Configured By attribute:

```
#define CLD_SCENES_ATTR_LAST_CONFIGURED_BY
```

Add this line to configure the maximum length of the Scene Name storage:

```
#define CLD_SCENES_MAX_SCENE_NAME_LENGTH            (16)
```

Add this line to configure the maximum number of scenes:

```
#define CLD_SCENES_MAX_NUMBER_OF_SCENES             (16)
```

Add this line to configure the maximum number of bytes available for scene storage:

```
#define CLD_SCENES_MAX_SCENE_STORAGE_BYTES          (20)
```

Further, enhanced functionality is available for the ZigBee Light Link (ZLL) profile and must be enabled as a compile-time option - for more information, refer to the *ZigBee Light Link User Guide (JN-UG-3091)*.

10. On/Off Cluster

This chapter describes the On/Off cluster which is defined in the ZCL.

The On/Off cluster has a Cluster ID of 0x0006.

10.1 Overview

The On/Off cluster allows a device to be put into the ‘on’ and ‘off’ states, or toggled between the two states. In the case of the ZigBee Light Link profile, the cluster also provides the following enhanced functionality:

- When switching off light(s) with an effect, saves the last light (attribute) settings to a global scene, ready to be re-used for the next switch-on from the global scene - see Section 10.4.2 and Section 10.5
- Allows light(s) to be switched on for a timed period (and then automatically switched off) - see Section 10.4.3

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 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 On/Off cluster are fully detailed in Section 10.9.

10.2 On/Off Cluster Structure and Attribute

The structure definition for the On/Off cluster is:

```
typedef struct
{
    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

} tsCLD_OnOff;
```

where:

- `bOnOff` is the on/off state of the device (TRUE = on, FALSE = off)
- `bGlobalSceneControl` is an optional ZLL attribute 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 ZLL attribute used to store the time, in tenths of a second, for which the lights will remain ‘on’ after a switch-on with ‘timed off’ (i.e. the time before starting the transition from the ‘on’ state to the ‘off’ state). The special values 0x0000 and 0xFFFF indicate the lamp must be maintained in the ‘on’ state indefinitely (no timed off)
- `u16OffWaitTime` is an optional ZLL attribute 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 9 and Chapter 8.

10.3 Initialisation

The function **eCLD_OnOffCreateOnOff()** is used to create an instance of the On/Off cluster. The function is generally called by the initialisation function for the host device.

Note: In the case of ZigBee Light Link, if the global scene is to be used to remember light settings then Scenes and Groups cluster instances must also be created - see Chapter 9 and Chapter 8.

10.4 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.

10.4.1 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 will be switched off and if the device is initially in the off state it will be switched on.

Note the following:

- For the ZigBee Light Link profile, a fourth option is available in the above function. This is to switch on with light settings retrieved for a global scene - for more information, refer to Section 10.5.
- For the Home Automation profile, if the Level Control cluster (see Chapter 12) 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 will result in a gradual transition 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 will result in a gradual transition from the ‘on’ level to the ‘off’ level over the time-interval specified by the attribute.

10.4.2 Switching Off Lights with Effect (ZLL Only)

In the case of the ZigBee Light Link profile, 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)

10.4.3 Switching On Timed Lights (ZLL Only)

In the case of the ZigBee Light Link profile, 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 will 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.

10.5 Saving Light Settings (ZLL Only)

In the case of the ZigBee Light Link profile, 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 subsequently switched on with the E_CLD_ONOFF_CMD_ON_RECALL_GLOBAL_SCENE option in **eCLD_OnOffCommandSend()**, the saved light settings are re-loaded. 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 particularly useful when the light levels are adjustable using the Level Control cluster (Chapter 12) and/or the light colours are adjustable using the Colour Control cluster (Chapter 17).

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.

10.6 Functions

The following On/Off cluster functions are provided in the NXP implementation of the ZCL:

Function	**Page**
eCLD_OnOffCreateOnOff	170
eCLD_OnOffCommandSend	172
eCLD_OnOffCommandOffWithEffectSend	174
eCLD_OnOffCommandOnWithTimedOffSend	176

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 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 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: This function must not be called for an endpoint on which a standard ZigBee 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 On/Off cluster function called in the application, and must be called after the stack has been started and after the application profile has been initialised.

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

The array declaration should be as follows:

```
uint8
au8AppOnOffClusterAttributeControlBits[CLD_ONOFF_MAX_NUMBER_OF_ATTRIBUTE];
```

The function will initialise the array elements to zero.

Parameters

psClusterInstance	Pointer to structure containing information about the cluster instance to be created (see Section 23.1.15). This structure will be 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 23.1.2). 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 will initialise 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 10.7.1)

Returns

E_ZCL_SUCCESS

E_ZCL_ERR_PARAMETER_NULL

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 is only available for the ZigBee Light Link profile and is described in Section 10.5).

The device receiving this message will generate a callback event on the endpoint on which the On/Off cluster was registered.

In the case of the Home Automation profile, if the Level Control cluster (see Chapter 12) 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 will result in a gradual transition 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 will result 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 will be 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 will be 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

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()**.

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 ZLL 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" on page 178.

The device receiving this message will generate 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 will be saved to a global scene, after which the attribute `bGlobalSceneControl` will be set to FALSE - for more details, refer to Section 10.5.

The function can be used only with the ZLL profile.

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 will be 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 will be 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 10.7.2)

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()**.

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 ZLL 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 light(s) cannot be switched on again.

The following functionality must be specified in the command payload:

- Time for which the light(s) must remain on
- Waiting time during which switched-off light(s) 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” on page 179.

The device receiving this message will generate a callback event on the endpoint on which the On/Off cluster was registered.

The function can be used only with the ZLL profile.

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 will be 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 will be 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 10.7.2)

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()**.

10.7 Structures

10.7.1 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.

10.7.2 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.

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

10.8 Enumerations

10.8.1 teCLD_OnOff_ClusterID

The following structure contains the enumerations used to identify the attributes of the On/Off cluster.

```
typedef enum PACK
{
    E_CLD_ONOFF_ATTR_ID_ONOFF                  = 0x0000,    /* Mandatory */
    E_CLD_ONOFF_ATTR_ID_GLOBAL_SCENE_CONTROL = 0x4000,    /* Optional */
    E_CLD_ONOFF_ATTR_ID_ON_TIME,                            /* Optional */
    E_CLD_ONOFF_ATTR_ID_OFF_WAIT_TIME,                      /* Optional */

} teCLD_OnOff_ClusterID;
```

10.8.2 teCLD_OOSC_SwitchType (On/Off Switch Types)

```
typedef enum PACK
{
    E_CLD_OOSC_TYPE_TOGGLE,
    E_CLD_OOSC_TYPE_MOMENTARY
} teCLD_OOSC_SwitchType;
```

10.8.3 teCLD_OOSC_SwitchAction (On/Off Switch Actions)

```
typedef enum PACK
{
    E_CLD_OOSC_ACTION_S2ON_S1OFF,
    E_CLD_OOSC_ACTION_S2OFF_S1ON,
    E_CLD_OOSC_ACTION_TOGGLE
} teCLD_OOSC_SwitchAction;
```

10.9 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.

Add this line to enable the optional Global Scene Control attribute (ZLL only):

```
#define CLD_ONOFF_ATTR_GLOBAL_SCENE_CONTROL
```

Add this line to enable the optional On Time attribute (ZLL only):

```
#define CLD_ONOFF_ATTR_ON_TIME
```

Add this line to enable the optional Off Wait Time attribute (ZLL only):

```
#define CLD_ONOFF_ATTR_OFF_WAIT_TIME
```

Further, enhanced functionality is available for the ZigBee Light Link (ZLL) profile and must be enabled as a compile-time option - for more information, refer to the *ZigBee Light Link User Guide (JN-UG-3091)*.

11. On/Off Switch Configuration Cluster

This chapter describes the On/Off Switch Configuration cluster which is defined in the ZCL.

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 10).

11.1 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 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 On/Off Switch Configuration cluster are fully detailed in Section 11.6.

11.2 On/Off Switch Config Cluster Structure and Attribute

The structure definition for the On/Off Switch Configuration cluster is:

```
typedef struct
{
    zenum8      eSwitchType;       /* Mandatory */
    zenum8      eSwitchActions;    /* Mandatory */
} 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 (e.g. 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 (e.g. a push-button which is pressed and then released)
- `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 fields of this structure, as detailed in Section 11.6.

11.3 Initialisation

The function **eCLD_OOSCCreateOnOffSwitchConfig()** is used to create an instance of the On/Off Switch Configuration cluster. The function is generally called by the initialisation function for the host device.

11.4 Functions

The following On/Off Switch Configuration cluster function is provided in the NXP implementation of the ZCL:

Function	**Page**
eCLD_OOSCCreateOnOffSwitchConfig	185

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.

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 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 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 On/Off Switch Configuration cluster function called in the application, and must be called after the stack has been started and after the application profile has been initialised.

Parameters

psClusterInstance	Pointer to structure containing information about the cluster instance to be created (see Section 23.1.15). This structure will be 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 23.1.2). 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 will initialise 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

11.5 Enumerations

11.5.1 teCLD_OOSC_ClusterID

The following structure contains the enumerations used to identify the attributes of the On/Off Switch Configuration cluster.

```
typedef enum PACK
{
    E_CLD_OOSC_ATTR_ID_SWITCH_TYPE       = 0x0000,   /* Mandatory */
    E_CLD_OOSC_ATTR_ID_SWITCH_ACTIONS    = 0x0010,   /* Mandatory */
} teCLD_OOSC_ClusterID;
```

11.6 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
```

The On/Off Switch Configuration cluster does not contain any optional functionality.

12. Level Control Cluster

This chapter describes the Level Control cluster which is defined in the ZCL.

The Level Control cluster has a Cluster ID of 0x0008.

12.1 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 normally be used with the On/Off cluster (see Chapter 10) 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 12.9.

12.2 Level Control Cluster Structure and Attributes

The structure definition for the Level Control cluster is shown below. The last three attributes are specific to the Home Automation (HA) profile.

```
typedef struct
{
    zuint8    u8CurrentLevel;

#ifdef CLD_LEVELCONTROL_ATTR_REMAINING_TIME
    zuint16   u16RemainingTime;
#endif

#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

} tsCLD_LevelControl;
```

where:

- `u8CurrentLevel` is the current level on the device
- `u16RemainingTime` is the time remaining (in tenths of a second) at the current level
- `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` will be set when an On command is received
- `u16OnTransitionTime` is an HA-specific 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 will also be used if `u16OnTransitionTime` is not enabled).
- `u16OffTransitionTime` is an HA-specific 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 will also be used if `u16OffTransitionTime` is not enabled).
- `u8DefaultMoveRate` is an HA-specific 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 12.7.2.2).

12.3 Initialisation

The function **eCLD_LevelControlCreateLevelControl()** is used to create an instance of the Level Control cluster. The function is generally called by the initialisation function for the host device.

12.4 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.

12.4.1 Changing Level

Three functions (see below) are provided for sending commands to change the current level on a device. These commands have the effect of modifying the 'current level' attribute of the Level Control cluster.

Each of these functions can be implemented in conjunction with the On/Off cluster. In this case:

- If the command increases the current level, the OnOff attribute of the On/Off cluster will be 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 will be 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 12.7.2.1). In the case of the ZigBee Light Link profile, the target level is interpreted as described in Section 12.5.1.

Move Command

The current level can be moved (up or down) at a specified rate using the function **eCLD_LevelControlCommandMoveCommandSend()**. The level will vary until stopped (see Section 12.4.2) or until the maximum or minimum level is reached. The direction and rate are specified in the command payload (see Section 12.7.2.2).

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 12.7.2.3).

12.4.2 Stopping a Level Change

A level change initiated using any of the functions referenced in Section 12.4.1 can be halted using the function **eCLD_LevelControlCommandStopCommandSend()** or **eCLD_LevelControlCommandStopWithOnOffCommandSend()**.

12.5 Issuing Local Commands

Some of the operations described in Section 12.4 that correspond to remote commands can also be performed locally, as described below.

12.5.1 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 will move towards 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 possible 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 will be 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 will be set to ‘off’.

12.5.2 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.

12.6 Functions

The following Level Control cluster functions are provided in the NXP implementation of the ZCL:

Function	Page
eCLD_LevelControlCreateLevelControl	194
eCLD_LevelControlSetLevel	196
eCLD_LevelControlGetLevel	197
eCLD_LevelControlCommandMoveToLevelCommandSend	198
eCLD_LevelControlCommandMoveCommandSend	200
eCLD_LevelControlCommandStepCommandSend	202
eCLD_LevelControlCommandStopCommandSend	204
eCLD_LevelControlCommandStopWithOnOffCommandSend	205

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.

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 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 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 Level Control cluster function called in the application, and must be called after the stack has been started and after the application profile has been initialised.

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

The array declaration should be as follows:

```
uint8 au8AppLevelControlClusterAttributeControlBits[
                                  CLD_LEVELCONTROL_MAX_NUMBER_OF_ATTRIBUTE];
```

The function will initialise the array elements to zero.

Parameters

psClusterInstance	Pointer to structure containing information about the cluster instance to be created (see Section 23.1.15). This structure will be 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 23.1.2). 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 will initialise 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 12.7.1)

Returns

E_ZCL_SUCCESS

E_ZCL_ERR_PARAMETER_NULL

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.

This operation can be performed in conjunction 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 will be 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 will be set to ‘off’.

Parameters

u8SourceEndPointId	Number of the local endpoint on which level is to be changed
u8Level	New level to be set, in the range 0x01 to 0x0FE
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()**.

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()**.

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 12.7.2).

The device receiving this message will generate a callback event on the endpoint on which the Level Control cluster was 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 will be 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 will be 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 will be set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

Parameters

u8SourceEndPointId	Number of the local endpoint through which 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 will be 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 will be 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 12.7.2)

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()**.

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 12.7.2).

If the current level reaches the maximum or minimum permissible level for the device, the level change will stop.

The device receiving this message will generate a callback event on the endpoint on which the Level Control cluster was 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 will be 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 will be 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 will be set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

Parameters

u8SourceEndPointId	Number of the local endpoint through which 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 will be 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 will be 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 12.7.2)

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()**.

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 12.7.2).

If the target level is above the maximum or below the minimum permissible level for the device, the stepped change will be limited to this level (and the transition time will be cut short).

The device receiving this message will generate a callback event on the endpoint on which the Level Control cluster was 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 will be 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 will be 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 will be set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

Parameters

u8SourceEndPointId	Number of the local endpoint through which 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 will be 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 will be 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 12.7.2)

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()**.

eCLD_LevelControlCommandStopCommandSend

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

Description

This function sends a Stop command to instruct a device to halt any transition to a new level.

The device receiving this message will generate a callback event on the endpoint on which the Level Control cluster was 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 will be set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

Parameters

u8SourceEndPointId	Number of the local endpoint through which 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 will be 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 will be 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()**.

eCLD_LevelControlCommandStopWithOnOffCommandSend

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

Description

This function sends a Stop with On/Off command to instruct a device to halt any transition to a new level.

The device receiving this message will generate a callback event on the endpoint on which the Level Control cluster was 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 will be set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

Parameters

u8SourceEndPointId	Number of the local endpoint through which 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 will be 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 will be 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()**.

12.7 Structures

12.7.1 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
{
    bool                                bUpdateAttributes;
    bool                                bWithOnOff;
    bool                                bRestoreLevelAfterOff;
    uint16                              u16RemainingTime;
    uint8                               u8TargetLevel;
    uint8                               u8PreviousLevel;
    tsZCL_ReceiveEventAddress           sReceiveEventAddress;
    tsZCL_CallBackEvent                 sCustomCallBackEvent;
    tsCLD_LevelControlCallBackMessage   sCallBackMessage;
} tsCLD_LevelControlCustomDataStructure;
```

The fields are for internal use and no knowledge of them is required.

12.7.2 Custom Command Payloads

The following structures contain the payloads for the Level Control cluster custom commands.

12.7.2.1 Move To Level Command Payload

```
typedef struct
{
    uint8                   u8Level;
    uint16                  u16TransitionTime;
} tsCLD_LevelControl_MoveToLevelCommandPayload;
```

where:

- `u8Level` is the target level
- `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 will change the level as fast as possible).

12.7.2.2 Move Command Payload

```
typedef struct
{
    uint8                   u8MoveMode;
    uint8                   u8Rate;
} 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 HA-specific `u8DefaultMoveRate` attribute instead - if this optional attribute is not present, the device will change the level as fast as possible)

12.7.2.3 Step Command Payload

```
typedef struct
{
    uint8                   u8StepMode;
    uint8                   u8StepSize;
    uint16                  u16TransitionTime;
} 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)

12.8 Enumerations

12.8.1 teCLD_LevelControl_ClusterID

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

```
typedef enum PACK
{
    E_CLD_LEVELCONTROL_ATTR_ID_CURRENT_LEVEL = 0x0000,
    E_CLD_LEVELCONTROL_ATTR_ID_REMAINING_TIME,
    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
} teCLD_LevelControl_ClusterID;
```

12.9 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 both of the following lines to the **zcl_options.h** file.

Add this line to enable the optional Remaining Time attribute:

```
#define CLD_LEVELCONTROL_ATTR_REMAINING_TIME
```

Add this line to enable the optional On/Off Transition Time attribute:

```
#define CLD_LEVELCONTROL_ATTR_ON_OFF_TRANSITION_TIME
```

Add this line to enable the optional On Level attribute:

```
#define CLD_LEVELCONTROL_ATTR_ON_LEVEL
```

Add this line to enable the optional HA-specific On Transition Time attribute:

```
#define CLD_LEVELCONTROL_ATTR_ON_TRANSITION_TIME
```

Add this line to enable the optional HA-specific Off Transition Time attribute:

```
#define CLD_LEVELCONTROL_ATTR_OFF_TRANSITION_TIME
```

Add this line to enable the optional HA-specific Default Move Rate attribute:

```
#define CLD_LEVELCONTROL_ATTR_DEFAULT_MOVE_RATE
```

13. 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-synchronise the ZigBee PRO devices.

The Time cluster has a Cluster ID of 0x000A.

This section also describes the maintenance of ‘ZCL time’.

13.1 Overview

The Time cluster is required in a ZigBee PRO network in which the constituent devices must be kept time-synchronised - for example, in a Smart Energy network in which certain devices must keep time with the ESP. In such a case, one device (e.g. the ESP) implements the Time cluster as a server and acts as the time-master for the network, while other devices in the network implement the Time cluster as a client and time-synchronise with the server.

Note that as for all clusters, the Time cluster is stored in a shared device structure (see Section 13.3) 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 the role of client or server must also be specified when creating the cluster instance). The compile-time options for the Time cluster are fully detailed in Section 13.10.

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

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

ZCL Time

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

13.2 Time Cluster Structure and Attributes

The Time cluster is contained in the following `tsCLD_Time` structure:

```
typedef struct
{
zutctime                utctTime; /* Mandatory */

zbmap8                  u8TimeStatus; /* Mandatory */

#ifdef E_CLD_TIME_ATTR_TIME_ZONE
zint32                  i32TimeZone;
#endif

#ifdef E_CLD_TIME_ATTR_DST_START
zuint32                 u32DstStart;
#endif

#ifdef E_CLD_TIME_ATTR_DST_END
zuint32                 u32DstEnd;
#endif

#ifdef E_CLD_TIME_ATTR_DST_SHIFT
zint32                  i32DstShift;
#endif

#ifdef E_CLD_TIME_ATTR_STANDARD_TIME
zuint32                 u32StandardTime;
#endif

#ifdef E_CLD_TIME_ATTR_LOCAL_TIME
zuint32                 u32LocalTime;
#endif

#ifdef E_CLD_TIME_ATTR_LAST_SET_TIME
zutctime                u32LastSetTime;
#endif

#ifdef E_CLD_TIME_ATTR_VALID_UNTIL_TIME
zutctime                u32ValidUntilTime;
#endif

} 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:

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

Table 7: u8TimeStatus Bitmap

* DST= Daylight Saving Time

Macros are provided for setting the individual bits of this bitmap:

- E_CLD_TM_TIME_STATUS_MASTER_MASK (bit 0)
- E_CLD_TM_TIME_STATUS_SYNCHRONIZED_MASK (bit 1)
- E_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.

Note: If required, 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 E_CLD_TIME_ATTR_TIME_ZONE (see Section 13.3.2).

13.3 Attribute Settings

13.3.1 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 13.5.

u8TimeStatus

This is a mandatory 8-bit attribute containing the bitmap detailed in Table 7 on page 213. 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 the utility company or from another external time-of-day source. Once the time has been obtained and set, the ‘Master’ bit should be set (to ‘1’).
- The ‘Synchronised’ bit must always be zero, as the 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 synchronised to the time-master).
- The ‘Master for Time Zone and DST’ bit must be set (to ‘1’) once the time-zone and Daylight Saving Time (DST) attributes (see below) have been correctly set for the device.

Macros are provided for setting the individual bits of the `u8TimeStatus` bitmap - for example, the macro E_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 13.2.

13.3.2 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 E_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 E_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 E_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 E_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 E_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 E_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 E_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 E_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.

13.4 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 initialised by the application using the function **vZCL_SetUTCTime()**.

The ZCL time is subsequently incremented from a local one-second timer provided by JenOS, as follows. On expiration of the JenOS timer, an event is generated (from the hardware/software timer that drives the JenOS timer), which causes JenOS to activate a ZCL user task. The event is initially handled by this task as described in Section 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 in a Smart Energy network).
2. The user task resumes the one-second timer using the JenOS function **OS_eContinueSWTimer()**.

13.4.1 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 JN5148 microcontroller normally uses its RC oscillator for timing purposes, which may not maintain the required accuracy (e.g. for Smart Energy). It is therefore recommended that a more accurate external crystal is used to time the sleep periods.

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

13.4.2 ZCL Time Synchronisation

The local ZCL time on a device can be synchronised with the time in a time-related cluster, such as Time, Price or Messaging. The ZCL time is considered to be synchronised following a call to **vZCL_SetUTCTime()**. The NXP implementation of the ZCL also provides the following functions relating to ZCL time synchronisation:

- **u32ZCL_GetUTCTime()** obtains the ZCL time (held locally).
- **bZCL_GetTimeHasBeenSynchronised()** determines whether the ZCL time on the device has been synchronised - 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 synchronised (for example, if there has been a problem in accessing the Time cluster server over a long period).

13.5 Time-Synchronisation of Devices

The devices in a ZigBee PRO network may need to be time-synchronised (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. In a Smart Energy network, the ESP normally acts as the time-master, since this device is linked to the utility company from where the master time is obtained.

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

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

On the time-master, these times are initialised by the local application using an external master time (e.g. using the current time from the Smart Energy utility company) and are subsequently maintained using a local one-second timer (see Section 13.5.1), as well as occasional re-synchronisations with external master time.

On all other devices, these times are initialised by the local application by synchronising with the time-master (see Section 13.5.2). The ZCL time is subsequently maintained using a local one-second timer and both times are occasionally re-synchronised with the time-master (see Section 13.5.3).

Synchronisation with the time-master is normally performed via the Time cluster (but, in a Smart Energy network, can alternatively be performed using a field of the Publish Price command).

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

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

The diagram in Figure 4 below provides an overview of the time initialisation and synchronisation processes described in the sub-sections that follow.

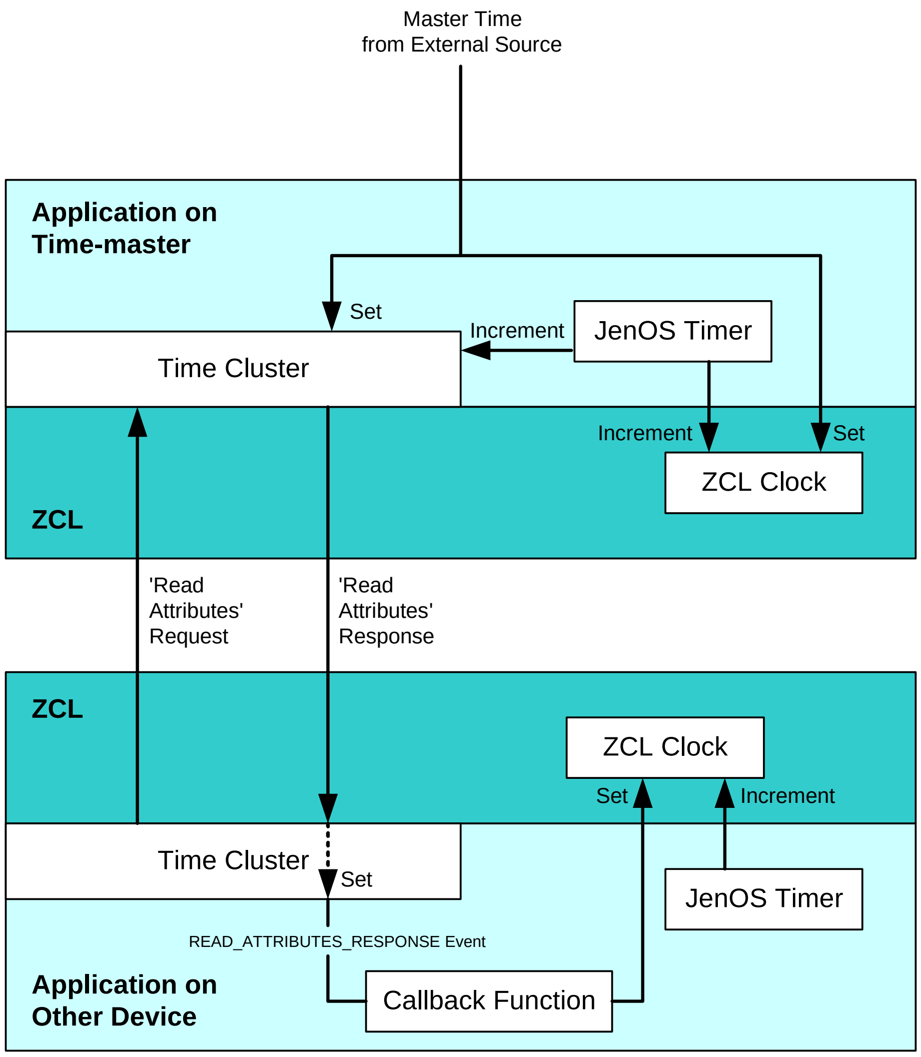


Figure 4: Time Initialisation and Synchronisation

13.5.1 Initialising and Maintaining Master Time

The time-master must initially obtain a master time from an external source - for example, in a Smart Energy network, the ESP initially obtains the current time (UTC) from the utility company. 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 13.3).

Note: The time-master can prevent other devices from attempting to read its Time cluster attributes before the time has been set - the initialisation 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 provided by JenOS, as follows. On expiration of the JenOS timer, an event is generated (from the hardware/software timer that drives the JenOS timer), which causes JenOS to activate a ZCL user task. The event is initially handled by this task as described in Section 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 using the JenOS function **OS_eContinueSWTimer()**.

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

Both the ZCL time and the `utctTime` attribute must also be updated by the application when an update of the master time is received (e.g. from the Smart Energy utility company).

13.5.2 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 JenOS to activate a ZCL user task. 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()**

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

Note: When a device attempts to time-synchronise with the 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 13.5.3.

13.5.3 Re-synchronisation of Devices

After the initialisation described in Section 13.5.2, the ZCL time must be updated by the application on each one-second tick of the local JenOS timer. The ZCL time is updated from the timer in the same way as described in Section 13.4.

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 on the ESP 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 13.5.2.

Notes for Smart Energy Networks

- If a Smart Energy device also implements the Price cluster, time re-synchronisation can be performed using the current time embedded in the Publish Price commands. However, these commands do not carry time-zone or daylight saving information. If such a command has not been received for an extended period of time, the device may need to initiate a time re-synchronisation with the ESP as described above.
- In order to avoid excessive re-synchronisation traffic across the network, the ZigBee Smart Energy specification states that "*time accuracy on client devices shall be within ±1 minute of the server device (ESP) per 24 hour period*". In addition, the specification demands that clock accuracy on the client devices "*never requires more than one time synchronization event per 24 hour period*". As a general rule, an application should initiate a time re-synchronisation if it has not received any communications that contain a time-stamp in the last 48 hours. However, in the case of a failed synchronisation (see Note in Section 13.5.2), a new attempt to synchronise can be made after a much shorter time, as this situation is only likely to occur when the ESP and the device have been powered around the same time.
- If the ESP receives a time update from the utility company then the ESP application must update its ZCL time and its time attribute.

13.6 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.

13.7 Functions

The following time-related functions are provided in the NXP implementation of the ZCL:

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"*

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. IPD of the SE profile) 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 application profile has been initialised.

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

The array declaration should be as follows:

```
uint8
au8AppTimeClusterAttributeControlBits[CLD_TIME_MAX_NUMBER_OF_ATTRIBUTE];
```

The function will initialise the array elements to zero.

Parameters

psClusterInstance	Pointer to structure containing information about the cluster instance to be created (see Section 23.1.15). This structure will be 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 23.1.2). 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 will initialise 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

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 13.2).

Parameters

u32UTCTime	The current time (UTC) to be set, in seconds

Returns

None

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

bZCL_GetTimeHasBeenSynchronised

```
bool_t bZCL_GetTimeHasBeenSynchronised(void);
```

Description

This function queries whether the ZCL time on the device has been synchronised.

The clock is considered to be unsynchronised at start-up and is synchronised following a call to **vZCL_SetUTCtime()**. The ZCL time must be synchronised before using the time-related functions of other SE clusters (e.g. the Price cluster).

Parameters

None

Returns

TRUE if the local ZCL time has been synchronised, otherwise FALSE

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.

For example, the application should call this function if it has been unable to maintain the ZCL time to within the one minute required by the Smart Energy specification - that is, if the application has been unable to call **vZCL_SetUTCTime()** for a long time.

Parameters

None

Returns

None

13.8 Return Codes

The time-related functions use the ZCL return codes defined in Section 24.2.

13.9 Enumerations

13.9.1 teCLD_TM_AttributeID

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

```
typedef enum PACK
{
    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;
```

13.10 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 E_CLD_TIME_ATTR_TIME_ZONE
```

Add this line to enable the optional DST Start attribute

```
#define E_CLD_TIME_ATTR_DST_START
```

Add this line to enable the optional DST End attribute

```
#define E_CLD_TIME_ATTR_DST_END
```

Add this line to enable the optional DST Shift attribute

```
#define E_CLD_TIME_ATTR_DST_SHIFT
```

Add this line to enable the optional Standard Time attribute

```
#define E_CLD_TIME_ATTR_STANDARD_TIME
```

Add this line to enable the optional Local Time attribute

```
#define E_CLD_TIME_ATTR_LOCAL_TIME
```

Note that some attributes must always be enabled together - for example, if daylight saving is to be implemented then E_CLD_TIME_ATTR_DST_START, E_CLD_TIME_ATTR_DST_END and E_CLD_TIME_ATTR_DST_SHIFT must all be included in the **zcl_options.h** file.

14. Binary Input (Basic) Cluster

This chapter describes the Binary Input (Basic) cluster which is defined in the ZCL, and which provides an interface for accessing a binary measurement and its associated characteristics.

The Binary Input (Basic) cluster has a Cluster ID of 0x000F.

14.1 Overview

The Binary Input (Basic) cluster provides an interface for accessing a binary measurement and its associated characteristics, and 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 **Binary_input_basic.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 14.5.

14.2 Binary Input (Basic) Structure and Attribute

The structure definition for the Binary Input (Basic) cluster is:

```
typedef struct
{

#ifdef CLD_BINARY_INPUT_BASIC_ATTR_ACTIVE_TEXT
        tsZCL_CharacterString       sActiveText;
        uint8                       au8ActiveText[16];
#endif

#ifdef CLD_BINARY_INPUT_BASIC_ATTR_DESCIRPTION
        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

} tsCLD_BinaryInputBasic;
```

- The following optional pair of attributes are used to store a human readable description of the active state of a binary input (e.g. “Window 3 open”):
  - `sActiveText` is a `tsZCL_CharacterString` structure (see Section 23.1.13) 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
- The following optional pair of attributes are used to store a human readable description of the usage of the binary input (e.g. “Window 3”):
  - `sDescription` is a `tsZCL_CharacterString` structure (see Section 23.1.13) 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 (e.g. “Window 3 closed”):
  - `sInactiveText` is a `tsZCL_CharacterString` structure (see Section 23.1.13) 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
- `bOutOfService` is an optional 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 a 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 `bPresentValue` in relation to the physical state of the input is determined by the setting of the `u8Polarity` attribute.
- `u8Reliability` is an optional attribute which indicates whether the value reported through `bPresentValue` is reliable and 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: Optional attribute `bOutOfService` is used and is TRUE • 0: Otherwise

Bits	Name	Description
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. Boiler Status). For a complete list of usages and the corresponding Index codes, refer to the attribute description in the ZCL Specification.
16-23	Type	Application usage domain. For the Basic Input cluster, this is 0x00 or 0x01, depending on the application usage. For lists of usages for each of these Type codes, refer to the attribute description in the ZCL Specification.
24-31	Group	The Cluster ID of the cluster that this attribute is part of. For the Binary Input (Basic) cluster, this is 0x000F.

14.3 Functions

The following Binary Input (Basic) cluster function is provided in the NXP implementation of the ZCL:

Function	Page
eCLD_BinaryInputBasicCreateBinaryInputBasic	237

The cluster attributes can be accessed using the general attribute read/write functions, as described in Section 2.2.

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 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 called after the stack has been started and after the application profile has been initialised.

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

The array declaration should be as follows:

```
uint8 au8AppBinaryInputBasicClusterAttributeControlBits
                                [CLD_BINARY_INPUT_BASIC_MAX_NUMBER_OF_ATTRIBUTE];
```

The function will initialise the array elements to zero.

Parameters

psClusterInstance	Pointer to structure containing information about the cluster instance to be created (see Section 23.1.15). This structure will be 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 23.1.2). 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 will initialise 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

14.4 Enumerations

14.4.1 teCLD_BinaryInputBasicCluster_AttrID

The following structure contains the enumerations used to identify the attributes of the Binary Input (Basic) cluster.

```
typedef enum PACK
{
    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;
```

14.4.2 teCLD_BinaryInputBasic_Polarity

The following structure contains the enumerations used to specify the value of the `u8Polarity` attribute (see Section 14.2).

```
typedef enum PACK
{
   E_CLD_ BINARY_INPUT_BASIC_POLARITY_NORMAL,
   E_CLD_ BINARY_INPUT_BASIC_POLARITY_REVERSE
}teCLD_BinaryInputBasic_Polarity
```

14.4.3 teCLD_BinaryInputBasic_Reliability

The following structure contains the enumerations used to report the value of the `u8Reliability` attribute (see Section 14.2).

```
typedef enum PACK
{
   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;
```

14.5 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 14.2) 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

15. 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.

15.1 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).

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 15.6.

Only server attributes are supported and all are optional. 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. The attribute values are set by the application but the application must ensure that these values are synchronised with the settings and NIB values for the ZigBee PRO stack.

15.2 Commissioning Cluster Structure and Attributes

The Commissioning cluster has only server attributes that are contained in the following `tsCLD_Commissioning` structure:

```
typedef struct
{
    /* Start-up attribute set (3.15.2.2) */
#ifdef     CLD_COMM_ATTR_SHORT_ADDRESS
    uint16              u16ShortAddress;
#endif

#ifdef     CLD_COMM_ATTR_EXTENED_PAN_ID
    zieeeaddress        u64ExtPanId;
#endif

#ifdef     CLD_COMM_ATTR_PAN_ID
    uint16              u16PANId;
#endif
```

```
#ifdef     CLD_COMM_ATTR_CHANNEL_MASK
    zbmap32               u32ChannelMask;
#endif


#ifdef     CLD_COMM_ATTR_PROTOCOL_VERSION
    uint8               u8ProtocolVersion;
#endif

#ifdef     CLD_COMM_ATTR_STACK_PROFILE
    uint8               u8StackProfile;
#endif

#ifdef     CLD_COMM_ATTR_START_UP_CONTROL
    zenum8               e8StartUpControl;
#endif

#ifdef     CLD_COMM_ATTR_TC_ADDR
    zieeeaddress         u64TcAddr;
#endif

#ifdef     CLD_COMM_ATTR_TC_MASTER_KEY
    tsZCL_Key         sTcMasterKey;
#endif

#ifdef     CLD_COMM_ATTR_NWK_KEY
    tsZCL_Key         sNwkKey;
#endif

#ifdef     CLD_COMM_ATTR_USE_INSECURE_JOIN
    bl_t         bUseInsecureJoin;
#endif

#ifdef     CLD_COMM_ATTR_PRE_CONFIG_LINK_KEY
    tsZCL_Key             sPreConfigLinkKey;
#endif

#ifdef     CLD_COMM_ATTR_NWK_KEY_SEQ_NO
    uint8         u8NwkKeySeqNo;
#endif

#ifdef     CLD_COMM_ATTR_NWK_KEY_TYPE
```

```
    zenum8         e8NwkKeyType;
#endif

#ifdef     CLD_COMM_ATTR_NWK_MANAGER_ADDR
    uint16         u16NwkManagerAddr;
#endif
    /* Join Parameters attribute set (3.15.2.2.2)*/
#ifdef     CLD_COMM_ATTR_SCAN_ATTEMPTS
    uint8         u8ScanAttempts;
#endif

#ifdef     CLD_COMM_ATTR_TIME_BW_SCANS
    uint16         u16TimeBwScans;
#endif

#ifdef     CLD_COMM_ATTR_REJOIN_INTERVAL
    uint16         u16RejoinInterval;
#endif

#ifdef     CLD_COMM_ATTR_MAX_REJOIN_INTERVAL
    uint16         u16MaxRejoinInterval;
#endif
    /* End Device Parameters attribute set (3.15.2.2.3)*/
#ifdef     CLD_COMM_ATTR_INDIRECT_POLL_RATE
    uint16         u16IndirectPollRate;
#endif

#ifdef     CLD_COMM_ATTR_PARENT_RETRY_THRSHLD
    uint8         u8ParentRetryThreshold;
#endif
    /* Concentrator Parameters attribute set (3.15.2.2.4)*/
#ifdef     CLD_COMM_ATTR_CONCENTRATOR_FLAG
    bl_t         bConcentratorFlag;
#endif

#ifdef     CLD_COMM_ATTR_CONCENTRATOR_RADIUS
    uint8         u8ConcentratorRadius;
#endif

#ifdef     CLD_COMM_ATTR_CONCENTRATOR_DISCVRY_TIME
    uint8         u8ConcentratorDiscoveryTime;
#endif
} tsCLD_Commissioning;
```

where:

‘Start-up Parameters’ Attribute Set

- `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 (thus, the latter value is needed for SE networks).
- `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.

‘Join Parameters’ Attribute Set

- `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.

‘End Device Parameters’ Attribute Set

- `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.

‘Concentrator Parameters’ Attribute Set

- `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.

Note: Memory is allocated at compile-time for all the Commissioning cluster attributes.

15.3 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 15.6) - for example, `u16ShortAddress` is enabled/disabled through the macro CLD_COMM_ATTR_SHORT_ADDRESS.

15.4 Functions

There are no Commissioning cluster functions.

15.5 Enumerations

15.5.1 teCLD_Commissioning_AttributeID

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

```
typedef enum PACK
{
    E_CLD_CMSNG_ATTR_ID_SHORT_ADDRESS               = 0x0000,
    E_CLD_CMSNG_ATTR_ID_EXT_PANID,
    E_CLD_CMSNG_ATTR_ID_PANID,
    E_CLD_CMSNG_ATTR_ID_CHANNEL_MASK,
    E_CLD_CMSNG_ATTR_ID_PROTOCOL_VERSION,
    E_CLD_CMSNG_ATTR_ID_STACK_PROFILE,
    E_CLD_CMSNG_ATTR_ID_STARTUP_CONTROl,
    E_CLD_CMSNG_ATTR_ID_TC_ADDR = 0x0010,
    E_CLD_CMSNG_ATTR_ID_TC_MASTER_KEY,
    E_CLD_CMSNG_ATTR_ID_NETWORK_KEY,
    E_CLD_CMSNG_ATTR_ID_USE_INSECURE_JOIN,
    E_CLD_CMSNG_ATTR_ID_PRECONFIG_LINK_KEY,
    E_CLD_CMSNG_ATTR_ID_NWK_KEY_SEQ_NO,
    E_CLD_CMSNG_ATTR_ID_NWK_KEY_TYPE,
    E_CLD_CMSNG_ATTR_ID_NWK_MANAGER_ADDR,
    E_CLD_CMSNG_ATTR_ID_SCAN_ATTEMPTS              = 0x0020,
    E_CLD_CMSNG_ATTR_ID_TIME_BW_SCANS,
    E_CLD_CMSNG_ATTR_ID_REJOIN_INTERVAL,
    E_CLD_CMSNG_ATTR_ID_MAX_REJOIN_INTERVAL,
    E_CLD_CMSNG_ATTR_ID_INDIRECT_POLL_RATE         = 0x0030,
    E_CLD_CMSNG_ATTR_ID_PARENT_RETRY_THRSHOLD,
    E_CLD_CMSNG_ATTR_ID_CONCENTRATOR_FLAG          = 0x0040,
    E_CLD_CMSNG_ATTR_ID_CONCENTRATOR_RADIUS,
    E_CLD_CMSNG_ATTR_ID_CONCENTRATOR_DISCVRY_TIME
} teCLD_Commissioning_AttributeID;
```

15.6 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
```

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 15.2 and Section 15.3):

```
#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
```

16. Door Lock Cluster

This chapter outlines the Door Lock cluster which is defined in the ZCL, and 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.

16.1 Overview

The Door Lock cluster is required in HA devices as indicated in the table below.

	Server-side	Client-side
Mandatory in...	Door Lock	Door Lock Controller
Optional in...		Remote Control

Table 8: Door Lock Cluster in HA Devices

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 16.8.

16.2 Door Lock Cluster Structure and Attributes

The Door Lock cluster is contained in the following `tsCLD_DoorLock` structure:

```
typedef struct
{

    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_SECURITY_LEVEL
    zuint8     u8ZigbeeSecurityLevel;
#endif

} 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
- `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 259.

16.3 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 `tsSM_CallBackMessage` structure. The possible command types are detailed below.

u8CommandId Enumeration	Description
E_CLD_DOOR_LOCK_CMD_LOCK	A lock request command has been received by the cluster server
E_CLD_DOOR_LOCK_CMD_UNLOCK	An unlock request command has been received by the cluster server

Table 9: Door Lock Command Types

16.4 Functions

The following Door Lock cluster functions are provided in the HA API:

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 application profile has been initialised.

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

The array declaration should be as follows:

```
uint8 au8AppDoorLockClusterAttributeControlBits[
                                               CLD_DOORLOCK_MAX_NUMBER_OF_ATTRIBUTE];
```

The function will initialise the array elements to zero.

Parameters

psClusterInstance	Pointer to structure containing information about the cluster instance to be created (see Section 23.1.15). This structure will be 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 23.1.2). 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 will initialise 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

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

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

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 will be set to match the TSN in the request, allowing an incoming response to be paired with a request.

Parameters

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

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 PRO Stack User Guide (JN-UG-3048)*.

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

16.5 Return Codes

The Door Lock cluster functions use the ZCL return codes defined in Section 24.2.

16.6 Enumerations

16.6.1 'Attribute ID' Enumerations

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

```
typedef enum PACK
{
    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;
```

16.6.2 'Lock State' Enumerations

The following enumerations are used to set the `eLockState` element in the Door Lock cluster structure `tsCLD_DoorLock`.

```
typedef enum PACK
{
    E_CLD_DOORLOCK_LOCK_STATE_NOT_FULLY_LOCKED = 0x00,
    E_CLD_DOORLOCK_LOCK_STATE_LOCK,
    E_CLD_DOORLOCK_LOCK_STATE_UNLOCK
} teCLD_DoorLock_LockState;
```

The above enumerations are described in the table below.

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

Table 10: 'Lock State' Enumerations

16.6.3 'Lock Type' Enumerations

The following enumerations are used to set the `eLockType` element in the Door Lock cluster structure `tsCLD_DoorLock`.

```
typedef enum PACK
{
    E_CLD_DOORLOCK_LOCK_TYPE_DEAD_BOLT = 0x00,
    E_CLD_DOORLOCK_LOCK_TYPE_MAGNETIC,
    E_CLD_DOORLOCK_LOCK_TYPE_OTHER
} teCLD_DoorLock_LockType;
```

The above enumerations are described in the table below.

Enumeration	Description
E_CLD_DOORLOCK_LOCK_TYPE_DEAD_BOLT	Dead bold lock
E_CLD_DOORLOCK_LOCK_TYPE_MAGNETIC	Magnetic lock
E_CLD_DOORLOCK_LOCK_TYPE_OTHER	Other type of lock

Table 11: 'Lock Type' Enumerations

16.6.4 ‘Door State’ Enumerations

The following enumerations are used to set the optional `eDoorState` element in the Door Lock cluster structure `tsCLD_DoorLock`.

```
typedef enum PACK
{
    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
} teCLD_DoorLock_DoorState;
```

The above enumerations are described in the table below.

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

Table 12: ‘Door State’ Enumerations

16.6.5 ‘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 PACK
{
    E_CLD_DOOR_LOCK_CMD_LOCK
    E_CLD_DOOR_LOCK_CMD_UNLOCK
} teCLD_DoorLock_CommandID;
```

The above enumerations are described in the table below.

Enumeration	Description
E_CLD_DOOR_LOCK_CMD_LOCK	A lock command
E_CLD_DOOR_LOCK_CMD_UNLOCK	An unlock command

Table 13: ‘Command ID’ Enumerations

16.7 Structures

16.7.1 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 16.7.2

16.7.2 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).

16.8 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

The optional attributes for the Door Lock cluster (see Section 16.2) are enabled by defining:

- CLD_DOOR_LOCK_ATTR_DOOR_STATE
- CLD_DOOR_LOCK_ATTR_NUMBER_OF_DOOR_OPEN_EVENTS
- CLD_DOOR_LOCK_ATTR_NUMBER_OF_DOOR_CLOSED_EVENTS
- CLD_DOOR_LOCK_ATTR_NUMBER_OF_MINUTES_DOOR_OPENED
- CLD_DOOR_LOCK_ZIGBEE_SECURITY_LEVEL

17. 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.

17.1 Overview

The Colour Control cluster is used to control the colour of a light.

Note 1: This cluster should normally be used with the Level Control cluster (see Chapter 12) and On/Off cluster (see Chapter 10). This is assumed to be the case in this description.

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 17.8.

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 Colour Control cluster are fully detailed in Section 17.8.

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

- Colour Information
- Defined Primaries Information
- Additional Defined Primaries Information
- Defined Colour Point Settings

There is also a set of of ‘enhanced’ attributes for the ZigBee Light Link profile.

17.2 Colour Control Cluster Structure and Attributes

The structure definition for the Colour Control cluster is:

```
typedef struct
{

/* 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
    zuint16                 u16ColourTemperature;
#endif

#ifdef CLD_COLOURCONTROL_ATTR_COLOUR_MODE
    zenum8                  u8ColourMode;
#endif


/* 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

/* ZLL enhanced attributes */
#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_PHY_MIN
    zuint16                 u16ColourTemperaturePhyMin;
#endif

#ifdef CLD_COLOURCONTROL_ATTR_COLOUR_TEMPERATURE_PHY_MAX
    zuint16                 u16ColourTemperaturePhyMax;
#endif
```

```
} tsCLD_ColourControl;
```

where:

'Colour Information' Attribute Set

- `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 normalised 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 normalised value of y is calculated using the following formula: y = `u16CurrentY`/65536.
- `u8DriftCompensation` indicates which mechanism, if any, is being used to compensate for colour/intensity drift over time. One of the following values is specified:

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 23.1.13) 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
- `u16ColourTemperature` is a scaled inverse of the current value of the colour temperature of the light, 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/`u16ColourTemperature`. This attribute is only valid when the attribute `u8ColorMode` 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:

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 (`u16ColourTemperature`)
0x03 - 0xFF	Reserved

'Defined Primaries Information' Attribute Set

- `u8NumberOfPrimaries` is the number of colour primaries implemented on the device, in the range 0-6 (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.

  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 normalised 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 normalised value of y is calculated using the following formula: y = `u16PrimaryNY`/65536.
- `u8PrimaryNIntensity` is a representation of the maximum intensity of colour primary 1, normalised 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.

'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 normalised 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

normalised 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 normalised 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 normalised 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, normalised 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 normalised 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 normalised 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, normalised 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 normalised 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 normalised 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, normalised such that the colour point with the highest relative intensity has the value 0xFE (the value 0xFF indicates an invalid value).

ZLL Enhanced Attributes

- `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:

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 (`u16ColourTemperature`)
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):

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`

Bits	Feature	Attributes
4	Colour temperature	`u16ColourTemperature` `u16ColourTemperaturePhyMin` `u16ColourTemperaturePhyMax`
5-15	Reserved	-

- `u16ColourTemperaturePhyMin` indicates the minimum value of the colour temperature attribute supported by the hardware.
- `u16ColourTemperaturePhyMax` indicates the maximum value of the colour temperature attribute supported by the hardware.

17.3 Initialisation

The function **eCLD_ColourControlCreateColourControl()** is used to create an instance of the Colour Control cluster. The function is generally called by the initialisation function for the host device.

17.4 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.

The Colour Control cluster includes some commands that are specific to the ZigBee Light Link (ZLL) profile. These commands relate to the ZLL 'enhanced' attributes of the cluster (see Section 17.2).

Note: In the case of ZLL, any 'Move to', 'Move' or 'Step' command that is currently in progress can be stopped at any time by calling the function:
eCLD_ColourControlCommandStopMoveStepCommandSend()

17.4.1 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 1: Hue can also be moved in conjunction with saturation, as described in Section 17.4.7.

Note 2: In the ZigBee Light Link (ZLL) profile, the ‘enhanced’ hue can be moved in similar ways, as described in Section 17.4.5.

17.4.2 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 17.4.7.

17.4.3 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 normalised 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()

17.4.4 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 ‘current colour temperature’ attribute (`u16ColourTemperature`) of the cluster. This attribute actually represents a scaled inverse 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/`u16ColourTemperature`

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 ‘current 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 ‘current 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 ‘current 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 ‘current 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 ‘current 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.

17.4.5 Controlling ‘Enhanced’ Hue (ZLL Only)

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 ZLL device that supports the Colour Control cluster, the hue can 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.

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)

Table 14: ‘Enhanced Current Hue’ Attribute Format

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:

$$\text{Enhanced hue} = H_i + (interp/255) \times (H_{i+1} - H_i)$$

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: 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 1: ‘Enhanced’ hue can also be moved in conjunction with saturation, as described in Section 17.4.7.

Note 2: The value of the ‘enhanced current hue’ attribute can be moved around a colour loop, as described in Section 17.4.6.

17.4.6 Controlling a Colour Loop (ZLL Only)

The colour of a ZLL 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 17.4.5 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.

17.4.7 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 (ZLL only)

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 (ZLL Only)

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_ColourControlCommandEnhancedMoveToHueAndSaturationCommand Send()

17.5 Functions

The following Colour Control cluster functions are provided in the NXP implementation of the ZCL:

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 will create 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 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 Colour Control cluster function called in the application, and must be called after the stack has been started and after the application profile has been initialised.

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

The array declaration should be as follows:

```
uint8 au8AppColourControlClusterAttributeControlBits[
                                    CLD_COLOURCONTROL_MAX_NUMBER_OF_ATTRIBUTE];
```

The function will initialise the array elements to zero.

Parameters

psClusterInstance	Pointer to structure containing information about the cluster instance to be created (see Section 23.1.15). This structure will be 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 23.1.2). 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 will initialise 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 17.6.1)

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()**.

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 17.6.2).

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 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 will be set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

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 will be 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 will be 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 17.6.2)

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()**.

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 17.6.2).

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 will be set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

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 will be 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 will be 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 17.6.2)

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()**.

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 17.6.2).

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 will be set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

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 will be 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 will be 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 17.6.2)

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()**.

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 17.6.2).

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 will be set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

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 will be 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 will be 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 17.6.2)

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()**.

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 17.6.2).

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 will be set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

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 will be 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 will be 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 17.6.2)

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()**.

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 17.6.2).

The device receiving this message will generate a callback event on the endpoint on which the Colour Control cluster was registered. he 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 will be set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

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 will be 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 will be 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 17.6.2)

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()**.

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 17.6.2).

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 will be set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

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 will be 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 will be 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 17.6.2)

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()**.

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 17.6.2).

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 will be set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

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 will be 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 will be 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 17.6.2)

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()**.

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 17.6.2 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 will be set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

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 will be 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 will be 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 17.6.2)

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()**.

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 17.6.2), 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 will be set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

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 will be 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 will be 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 17.6.2)

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()**.

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 ZLL device to move its ‘enhanced current hue’ attribute to a target hue value in a continuous manner within a given time. The function can be used only with the ZLL profile. The enhanced hue value, direction and transition time are specified in the payload of the command (see Section 17.6.2). 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 will be set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

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 will be 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 will be 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 17.6.2)

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()**.

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 ZLL device to move its ‘enhanced current hue’ attribute value in a given direction at a specified rate for an indefinite time. The function can be used only with the ZLL profile. The direction and rate are specified in the payload of the command (see Section 17.6.2). 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 will be set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

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 will be 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 will be 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 17.6.2)

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()**.

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 ZLL device to increase or decrease its 'enhanced current hue' attribute by a specified 'step' value in a continuous manner within a given time. The function can be used only with the ZLL profile. The step size, direction and transition time are specified in the payload of the command (see Section 17.6.2). 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 will be set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

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 will be 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 will be 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 17.6.2)

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()**.

eCLD_ColourControlCommandEnhancedMoveToHueAndSaturationCommandSend

```
teZCL_Status
eCLD_ColourControlCommandEnhancedMoveToHueAndSaturationCommand
Send(
        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 ZLL device to move its 'enhanced current hue' and 'current saturation' attributes to target values in a continuous manner within a given time. The function can be used only with the ZLL profile. The enhanced hue value, saturation value and transition time are specified in the payload of the command (see Section 17.6.2). 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 will be set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

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 will be 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 will be 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 17.6.2)

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()**.

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 ZLL device to configure the movement of the ‘enhanced current hue’ attribute value around the colour loop corresponding to the CIE colour ‘triangle’. The function can be used only with the ZLL profile. 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 17.6.2). 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 will be set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

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 will be 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 will be 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 17.6.2)

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()**.

eCLD_ColourControlCommandStopMoveStepCommandSend

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

Description

This function sends a Stop Move Step command to instruct a ZLL device to stop a ‘Move to’, ‘Move’ or ‘Step’ command that is currently in progress. The function can be used only with the ZLL profile.

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 will be set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

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 will be 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 will be 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()**.

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 ‘colour temperature’ attribute to a target value in a continuous manner within a given time. The attribute value is actually a scaled reciprocal of colour temperature, as indicated in Section 17.4.4. The target attribute value, direction and transition time are specified in the payload of the command (see Section 17.6.2).

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 ‘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 will be set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

This function can only be used when the ‘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 will be 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 will be 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 17.6.2)

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()**.

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 ZLL device to move its ‘colour temperature’ attribute value in a given direction at a specified rate. The attribute value is actually a scaled reciprocal of colour temperature, as indicated in Section 17.4.4. The direction and rate are specified in the payload of the command (see Section 17.6.2). Maximum and minimum attribute values for the movement are also specified in the payload. The function can be used only with the ZLL profile.

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 ‘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 will be set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

This function can only be used when the ‘colour temperature’ attribute is enabled in the Colour Control cluster, as well as the ‘colour temperature maximum’ and ‘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 will be 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 will be 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 17.6.2)

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()**.

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 ZLL device to increase or decrease its ‘colour temperature’ attribute by a specified ‘step’ value in a continuous manner within a given time. The attribute value is actually a scaled reciprocal of colour temperature, as indicated in Section 17.4.4. The step size, direction and transition time are specified in the payload of the command (see Section 17.6.2). Maximum and minimum attribute values for the movement are also specified in the payload. The function can be used only with the ZLL profile.

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 ‘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 will be set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

This function can only be used when the ‘colour temperature’ attribute is enabled in the Colour Control cluster, as well as the ‘colour temperature maximum’ and ‘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 will be 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 will be 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 17.6.2)

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()**.

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 ZLL device on the specified (local) endpoint in terms of the Red (R), Green (G) and Blue (B) components. The function can be used only with the ZLL profile.

Parameters

u8SourceEndPointId	Number of local endpoint on which the ZLL 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()**.

17.6 Structures

17.6.1 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.

17.6.2 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;
} 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).

Step Hue Command Payload

```
typedef struct
{
    teCLD_ColourControl_StepMode    eMode;
    uint8                           u8StepSize;
    uint8                           u8TransitionTime;
} 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
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.

Move To Saturation Command Payload

```
typedef struct
{
    uint8       u8Saturation;
    uint16      u16TransitionTime;
} 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.

Move Saturation Command Payload

```
typedef struct
{
    teCLD_ColourControl_MoveMode    eMode;
    uint8                           u8Rate;
} 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

eMode	Action/Direction
0x02	Reserved
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).

Step Saturation Command Payload

```
typedef struct
{
    teCLD_ColourControl_StepMode    eMode;
    uint8                           u8StepSize;
    uint8                           u8TransitionTime;
} 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.

Move To Hue And Saturation Command Payload

```
typedef struct
{
    uint8       u8Hue;
    uint8       u8Saturation;
    uint16      u16TransitionTime;
} 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.

Move To Colour Command Payload

```
typedef struct
{
    uint16    u16ColourX;
    uint16    u16ColourY;
    uint16    u16TransitionTime;
} 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.

Move Colour Command Payload

```
typedef struct
{
    int16     i16RateX;
    int16     i16RateY;
} 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).

Step Colour Command Payload

```
typedef struct
{
    int16                          i16StepX;
    int16                          i16StepY;
    uint16                         u16TransitionTime;
} 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.

Move To Colour Temperature Command Payload

```
typedef struct
{
    uint16      u16ColourTemperature;
    uint16      u16TransitionTime;
} tsCLD_ColourControl_MoveToColourTemperatureCommandPayload;
```

where:

- `u16ColourTemperature` is the target value of the colour temperature attribute `u16ColourTemperature` (this value is a scaled inverse of colour temperature - for details, refer to the attribute description in Section 17.2).
- `u16TransitionTime` is the time period, in tenths of a second, over which the change in colour temperature should be implemented.

Move Colour Temperature Command Payload

```
typedef struct
{
    teCLD_ColourControl_MoveMode   eMode;
    uint16                         u16Rate;
    uint16                         u16ColourTemperatureMin;
    uint16                         u16ColourTemperatureMax;
} tsCLD_ColourControl_MoveColourTemperatureCommandPayload;
```

where:

- `eMode` indicates the required action and/or direction of the change in the colour temperature attribute value:

eMode	Action/Direction
0x00	Stop existing movement in colour temperature
0x01	Start increasing colour temperature attribute value
0x02	Reserved
0x03	Start decreasing colour temperature attribute value
0x04 – 0xFF	Reserved

- `u16Rate` is the required rate of movement in colour temperature steps per second (a step is one unit of the colour temperature attribute for the device).
- `u16ColourTemperatureMin` is the lower limit for the colour temperature attribute during the operation resulting from this command.
- `u16ColourTemperatureMax` is the upper limit for the colour temperature attribute during the operation resulting from this command.

Step Colour Temperature Command Payload

```
typedef struct
{
    teCLD_ColourControl_StepMode eMode;
    uint16                       u16StepSize;
    uint16                       u16TransitionTime;
    uint16                       u16ColourTemperatureMin;
    uint16                       u16ColourTemperatureMax;
} tsCLD_ColourControl_StepColourTemperatureCommandPayload;
```

where:

- `eMode` indicates the required direction of the change in the colour temperature attribute value:

eMode	Action/Direction
0x00	Reserved
0x01	Increase colour temperature attribute value
0x02	Reserved
0x03	Decrease colour temperature attribute value
0x04 – 0xFF	Reserved

- `u16StepSize` is the amount by which the 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 colour temperature attribute should be implemented.
- `u16ColourTemperatureMin` is the lower limit for the colour temperature attribute during the operation resulting from this command.
- `u16ColourTemperatureMax` is the upper limit for the colour temperature attribute during the operation resulting from this command.

Enhanced Move To Hue Command Payload

```
typedef struct
{
    uint16                          u16EnhancedHue;
    teCLD_ColourControl_Direction   eDirection;
    uint16                          u16TransitionTime;
} 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 17.2.
- `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.

Enhanced Move Hue Command Payload

```
typedef struct
{
    teCLD_ColourControl_MoveMode       eMode;
    uint16                             u16Rate;
} 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).

Enhanced Step Hue Command Payload

```
typedef struct
{
    teCLD_ColourControl_StepMode        eMode;
    uint16                              u16StepSize;
    uint16                              u16TransitionTime;
} tsCLD_ColourControl_EnhancedStepHueCommandPayload;
```

where:

- eMode indicates the required direction of the change in hue:

eMode	Action/Direction
0x00	Reserved
0x01	Increase in hue
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 17.2.
- u8TransitionTime is the time period, in tenths of a second, over which the change in hue should be implemented.

Enhanced Move To Hue And Saturation Command Payload

```
typedef struct
{
    uint16      u16EnhancedHue;
    uint8       u8Saturation;
    uint16      u16TransitionTime;
}
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 17.2.
- 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.

Colour Loop Set Command Payload

```
typedef struct
{
    uint8                                   u8UpdateFlags;
    teCLD_ColourControl_LoopAction          eAction;
    teCLD_ColourControl_LoopDirection       eDirection;
    uint16                                  u16Time;
    uint16                                  u16StartHue;
} 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`).

17.7 Enumerations

17.7.1 teCLD_ColourControl_ClusterID

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

```
typedef enum PACK
{
    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,
    E_CLD_COLOURCONTROL_ATTR_COLOUR_MODE,
    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_PHY_MIN,
    E_CLD_COLOURCONTROL_ATTR_COLOUR_TEMPERATURE_PHY_MAX
} teCLD_ColourControl_ClusterID;
```

17.8 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
```

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_CURRENT_HUE
  - CLD_COLOURCONTROL_ATTR_CURRENT_SATURATION

  - CLD_COLOURCONTROL_ATTR_REMAINING_TIME
  - CLD_COLOURCONTROL_ATTR_DRIFT_COMPENSATION
  - CLD_COLOURCONTROL_ATTR_COMPENSATION_TEXT
  - CLD_COLOURCONTROL_ATTR_COLOUR_TEMPERATURE
  - CLD_COLOURCONTROL_ATTR_COLOUR_MODE
- For optional attributes from the ‘Defined Primaries Information’ attribute set:
  - CLD_COLOURCONTROL_ATTR_NUMBER_OF_PRIMARIES
  - CLD_COLOURCONTROL_ATTR_PRIMARY_1_X
  - CLD_COLOURCONTROL_ATTR_PRIMARY_1_Y
  - CLD_COLOURCONTROL_ATTR_PRIMARY_1_INTENSITY
  - CLD_COLOURCONTROL_ATTR_PRIMARY_2_X
  - CLD_COLOURCONTROL_ATTR_PRIMARY_2_Y
  - CLD_COLOURCONTROL_ATTR_PRIMARY_2_INTENSITY
  - CLD_COLOURCONTROL_ATTR_PRIMARY_3_X
  - CLD_COLOURCONTROL_ATTR_PRIMARY_3_Y
  - CLD_COLOURCONTROL_ATTR_PRIMARY_3_INTENSITY
- For optional attributes from the ‘Additional Defined Primaries Information’ attribute set:
  - CLD_COLOURCONTROL_ATTR_PRIMARY_4_X
  - CLD_COLOURCONTROL_ATTR_PRIMARY_4_Y
  - CLD_COLOURCONTROL_ATTR_PRIMARY_4_INTENSITY
  - CLD_COLOURCONTROL_ATTR_PRIMARY_5_X
  - CLD_COLOURCONTROL_ATTR_PRIMARY_5_Y
  - CLD_COLOURCONTROL_ATTR_PRIMARY_5_INTENSITY
  - CLD_COLOURCONTROL_ATTR_PRIMARY_6_X
  - CLD_COLOURCONTROL_ATTR_PRIMARY_6_Y
  - CLD_COLOURCONTROL_ATTR_PRIMARY_6_INTENSITY
- 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 ZLL enhanced attributes:

- CLD_COLOURCONTROL_ATTR_ENHANCED_CURRENT_HUE
- CLD_COLOURCONTROL_ATTR_ENHANCED_COLOUR_MODE
- CLD_COLOURCONTROL_ATTR_COLOUR_LOOP_ACTIVE
- CLD_COLOURCONTROL_ATTR_COLOUR_LOOP_DIRECTION
- CLD_COLOURCONTROL_ATTR_COLOUR_LOOP_TIME
- CLD_COLOURCONTROL_ATTR_COLOUR_LOOP_START_ENHANCED_HUE
- CLD_COLOURCONTROL_ATTR_COLOUR_LOOP_STORED_ENHANCED_HUE
- CLD_COLOURCONTROL_ATTR_COLOUR_CAPABILITIES
- CLD_COLOURCONTROL_ATTR_COLOUR_TEMPERATURE_PHY_MIN
- CLD_COLOURCONTROL_ATTR_COLOUR_TEMPERATURE_PHY_MAX

Further, enhanced functionality is available for the ZigBee Light Link (ZLL) profile and must be enabled as a compile-time option - for more information, refer to the *ZigBee Light Link User Guide (JN-UG-3091)*.

18. Illuminance Measurement Cluster

This chapter describes the Illuminance Measurement cluster which is defined in the ZCL and provides an interface to a light sensor which is able to make illuminance measurements.

The Illuminance Measurement cluster has a Cluster ID of 0x0400.

18.1 Overview

The Illuminance Measurement cluster provides an interface to an illuminance measuring device, allowing the configuration of illuminance measuring and the reporting of illuminance 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 18.5.

18.2 Illuminance Measurement Structure and Attributes

The structure definition for the Illuminance Measurement cluster is:

```
typedef struct
{
    zuint16                u16MeasuredValue;
    zuint16                u16MinMeasuredValue;
    zuint16                u16MaxMeasuredValue;

#ifdef E_CLD_ILLMEAS_ATTR_TOLERANCE
    zuint16                u16Tolerance;
#endif

#ifdef E_CLD_ILLMEAS_ATTR_LIGHT_SENSOR_TYPE
    zenum8                 eLightSensorType;
#endif

} 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 is used to indicated 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 is used to indicated 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 will be in the range (`u16MeasuredValue` – `u16Tolerance`) to (`u16MeasuredValue` + `u16Tolerance`) .
- `eLightSensorType` is an optional attribute which 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

18.3 Functions

The following Illuminance Measurement cluster function is provided in the NXP implementation of the ZCL:

Function	**Page**
eCLD_IlluminanceMeasurementCreateIlluminanceMeasurement	342

The cluster attributes can be accessed using the general attribute read/write functions, as described in Section 2.2.

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 will create 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 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 called after the stack has been started and after the application profile has been initialised.

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

The array declaration should be as follows:

```
uint8 au8AppIlluminance MeasurementClusterAttributeControlBits[
                                         CLD_ILLMEAS_MAX_NUMBER_OF_ATTRIBUTE];
```

The function will initialise the array elements to zero.

Parameters

psClusterInstance	Pointer to structure containing information about the cluster instance to be created (see Section 23.1.15). This structure will be 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 23.1.2). 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 will initialise 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

18.4 Enumerations

18.4.1 teCLD_IM_ClusterID

The following structure contains the enumeration used to identify the attributes of the Illuminance Measurement cluster.

```
typedef enum PACK
{
    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;
```

18.5 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 or both, it is necessary to add one of the following to the same file:

```
#define ILLUMINANCE_MEASUREMENT_CLIENT
#define ILLUMINANCE_MEASUREMENT_SERVER
```

Optional Attributes

The optional attributes for the Illuminance Measurement cluster (see Section 18.2) are enabled by defining:

- E_CLD_ILLMEAS_ATTR_TOLERANCE
- E_CLD_ILLMEAS_ATTR_LIGHT_SENSOR_TYPE

19. Occupancy Sensing Cluster

This chapter describes the Occupancy Sensing cluster which is defined in the ZCL and provides an interface to an occupancy sensor.

The Occupancy Sensing cluster has a Cluster ID of 0x0406.

19.1 Overview

The Occupancy Sensing cluster provides an interface to an occupany sensor, allowing the configuration of occupany sensing and the reporting of the occupancy 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 19.5.

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 19.2) 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 2.2.

19.2 Occupancy Sensing Structure and Attributes

The structure definition for the Occupancy Sensing cluster is:

```
typedef struct
{
    zbmap8    u8Occupancy;
    zenum8    eOccupancySensorType;

#ifdef E_CLD_OS_ATTR_PIR_OCCUPIED_TO_UNOCCUPIED_DELAY
    zuint16   u16PIROccupiedToUnoccupiedDelay;
#endif

#ifdef E_CLD_OS_ATTR_PIR_UNOCCUPIED_TO_OCCUPIED_DELAY
    zuint8    u8PIRUnoccupiedToOccupiedDelay;
#endif

#ifdef E_CLD_OS_ATTR_PIR_UNOCCUPIED_TO_OCCUPIED_THRESHOLD
    zuint8     u8PIRUnoccupiedToOccupiedThreshold;
#endif

#ifdef E_CLD_OS_ATTR_ULTRASONIC_OCCUPIED_TO_UNOCCUPIED_DELAY
    zuint16    u16UltrasonicOccupiedToUnoccupiedDelay;
#endif

#ifdef E_CLD_OS_ATTR_ULTRASONIC_UNOCCUPIED_TO_OCCUPIED_DELAY
    zuint8     u8UltrasonicUnoccupiedToOccupiedDelay;
#endif

#ifdef E_CLD_OS_ATTR_ULTRASONIC_UNOCCUPIED_TO_OCCUPIED_THRESHOLD
    zuint8     u8UltrasonicUnoccupiedToOccupiedThreshold;
#endif
} 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 (e.g. in a corridor).

19.3 Functions

The following Occupancy Sensing cluster function is provided in the NXP implementation of the ZCL:

Function	Page
eCLD_OccupancySensingCreateOccupancySensing	349

The cluster attributes can be accessed using the general attribute read/write functions, as described in Section 2.2. 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 19.2).

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 will create 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 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 called after the stack has been started and after the application profile has been initialised.

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

The array declaration should be as follows:

```
uint8 au8AppOccupancySensingClusterAttributeControlBits[
                                        CLD_OS_MAX_NUMBER_OF_ATTRIBUTE];
```

The function will initialise the array elements to zero.

Parameters

psClusterInstance	Pointer to structure containing information about the cluster instance to be created (see Section 23.1.15). This structure will be 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 23.1.2). 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 will initialise 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 Enumerations

19.4.1 teCLD_OS_ClusterID

The following structure contains the enumeration used to identify the attributes of the Occupancy Sensing cluster.

```
typedef enum PACK
{
    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;
```

19.5 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

The optional attributes for the Occupancy Sensing cluster (see Section 19.2) are enabled by defining:

- E_CLD_OS_ATTR_PIR_OCCUPIED_TO_UNOCCUPIED_DELAY
- E_CLD_OS_ATTR_PIR_UNOCCUPIED_TO_OCCUPIED_DELAY
- E_CLD_OS_ATTR_PIR_UNOCCUPIED_TO_OCCUPIED_THRESHOLD
- E_CLD_OS_ATTR_ULTRASONIC_OCCUPIED_TO_UNOCCUPIED_DELAY
- E_CLD_OS_ATTR_ULTRASONIC_UNOCCUPIED_TO_OCCUPIED_DELAY
- E_CLD_OS_ATTR_ULTRASONIC_UNOCCUPIED_TO_OCCUPIED_THRESHOLD

20. OTA Upgrade Cluster

This chapter describes the Over-The-Air (OTA) Upgrade cluster. This cluster is not officially a part of the ZCL but is described in this manual as it can be included in any ZigBee application profile (but most notably Smart Energy).

The OTA Upgrade cluster has a Cluster ID of 0x0019.

Note 1: ZigBee PRO can be run on the JN516x and JN5148-Z01 microcontrollers. The JN516x devices have internal Flash memory whilst the JN5148-Z01 device requires an external Flash memory device. As a result, the bootloaders are different - refer to Appendix C. A JN516x device also requires an external Flash memory device to participate in OTA upgrades.

Note 2: This chapter largely assumes that the ZigBee PRO network consists of nodes which contain only one processor - a JN51xx microcontroller. However, the OTA Upgrade cluster can also be used with dual-processor nodes (containing a JN51xx device and a co-processor), as described in Appendix D.

20.1 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 (e.g. in the case of a Smart Energy system, from the utility company via the backhaul network).

An application that uses the OTA Upgrade cluster must include the header files **zcl_options.h**, **OTA.h** and **ovly.h** (the overlay header file, **ovly.h**, is only applicable to the JN5148 device).

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 20.12.

20.2 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:

```
const tsZCL_AttributeDefinition asOTAClusterAttributeDefinitions[] = {

/* ZigBee Cluster Library Version */
{E_CLD_OTA_ATTR_UPGRADE_SERVER_ID, E_ZCL_AF_RD | E_ZCL_AF_CA,
E_ZCL_IEEE_ADDR, (uint16)(&((tsCLD_AS_Ota*)(0))->u64UgradeServerID),0},
/* Mandatory */

#ifdef OTA_CLD_ATTR_FILE_OFFSET
{E_CLD_OTA_ATTR_FILE_OFFSET, E_ZCL_AF_RD | E_ZCL_AF_CA, E_ZCL_UINT32,
(uint16)(&((tsCLD_AS_Ota*)(0))->u32FileOffset), 0}, /* Optional */
#endif

#ifdef OTA_CLD_ATTR_CURRENT_FILE_VERSION
{E_CLD_OTA_ATTR_CURRENT_FILE_VERSION, E_ZCL_AF_RD | E_ZCL_AF_CA,
E_ZCL_UINT32,(uint16)(&((tsCLD_AS_Ota*)(0))->u32CurrentFileVersion),0},
/* Optional */
#endif

#ifdef OTA_CLD_ATTR_CURRENT_ZIGBEE_STACK_VERSION
{E_CLD_OTA_ATTR_CURRENT_ZIGBEE_STACK_VERSION, E_ZCL_AF_RD | E_ZCL_AF_CA,
E_ZCL_UINT16, (uint16)(&((tsCLD_AS_Ota*)(0))->u16CurrentStackVersion),
0}, /* Optional */
#endif

#ifdef OTA_CLD_ATTR_DOWNLOADED_FILE_VERSION
{E_CLD_OTA_ATTR_DOWNLOADED_FILE_VERSION, E_ZCL_AF_RD | E_ZCL_AF_CA,
E_ZCL_UINT32, (uint16)(&((tsCLD_AS_Ota*)(0))->u32DownloadedFileVersion),
0}, /* Optional */
#endif

#ifdef OTA_CLD_ATTR_DOWNLOADED_ZIGBEE_STACK_VERSION
{E_CLD_OTA_ATTR_DOWNLOADED_ZIGBEE_STACK_VERSION, E_ZCL_AF_RD |
E_ZCL_AF_CA, E_ZCL_UINT16, (uint16)(&((tsCLD_AS_Ota*)(0))-
>u16DownloadedStackVersion), 0}, /* Optional */
#endif

{E_CLD_OTA_ATTR_IMAGE_UPGRADE_STATUS, E_ZCL_AF_RD | E_ZCL_AF_CA,
E_ZCL_ENUM8, (uint16)(&((tsCLD_AS_Ota*)(0))->u8ImageUpgradeStatus), 0},
/* Mandatory */

#ifdef OTA_CLD_ATTR_MANF_ID
{E_CLD_OTA_ATTR_MANF_ID, E_ZCL_AF_RD | E_ZCL_AF_CA, E_ZCL_UINT16,
(uint16)(&((tsCLD_AS_Ota*)(0))->u16ManfId), 0}, /* Optional */
#endif

#ifdef OTA_CLD_ATTR_IMAGE_TYPE
{E_CLD_OTA_ATTR_IMAGE_TYPE, E_ZCL_AF_RD | E_ZCL_AF_CA, E_ZCL_UINT16,
(uint16)(&((tsCLD_AS_Ota*)(0))->u16ImageType), 0}, /* Optional */
#endif

#ifdef OTA_CLD_ATTR_REQUEST_DELAY
{E_CLD_OTA_ATTR_REQUEST_DELAY, E_ZCL_AF_RD | E_ZCL_AF_CA, E_ZCL_UINT16,
(uint16)(&((tsCLD_AS_Ota*)(0))->u16MinBlockRequestDelay), 0},
/* Optional */
#endif
};
```

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 (external) 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, signature verified and image saved to memory
0x03	Waiting to upgrade	Waiting for instruction from server to upgrade from the saved image
0x04	Count down	Client has been instructed by server 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 milliseconds, 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 0-600 ms. 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.

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 20.12).

20.3 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 (e.g. 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.

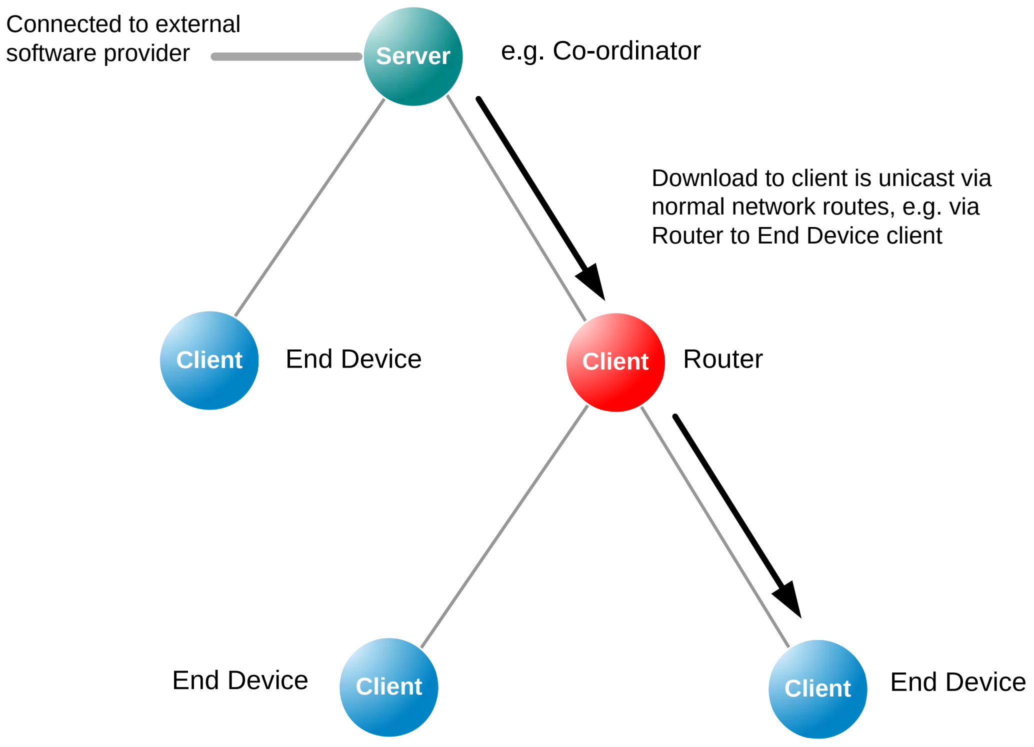

Connected to external software provider
Server
e.g. Co-ordinator
Download to client is unicast via normal network routes, e.g. via Router to End Device client
Client
End Device
Client
Router
End Device
Client
Client
End Device

Figure 5: OTA Download Example

The upgrade application is downloaded into an external Flash memory device which is attached to the JN51xx device on the client node. The application is then loaded into RAM or internal Flash memory (JN5148-Z01 or JN516x respectively) and executed. Note that the final sector of external Flash memory should normally be reserved for persistent data storage - for example, in an 8-sector 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.

20.3.1 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 - for example, in a Smart Energy network, the server is normally the Energy Service Portal (ESP) device which is connected to the utility company via a backhaul network. The server would also usually be the Co-ordinator 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 20.3.2 below).

20.3.2 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 will not always be 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.

20.4 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**, **OTA.h** and **ovly.h** (the overlay header file is only applicable to a JN514x device)
- Inclusion of the relevant #defines in the file **zcl_options.h**, as described in Section 20.12
- Specific application initialisation requirements, as outlined in Section 20.5
- Use of overlays (paging), as outlined in Section 20.7.4 (JN514x only)
- Use of the JenOS Persistent Data Manager (PDM) to preserve context data, as outlined in Section 20.7.5
- Use of a JenOS mutex to protect accesses to Flash memory via the SPI bus, as outlined in Section 20.7.6
- Organisation of Flash memory, as outlined in Section 20.7.7
- For a Smart Energy system, compulsory use of the Key Establishment cluster for security, as outlined in Section 20.7.8
- Optionally, a signature may be appended to an upgrade image, as described in Section 20.7.9
- When using a non-SE profile (such as Home Automation), it is necessary to remove references to the Certicom security certificate, as indicated in Section 20.12

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 20.5 and Section 20.7.

20.5 Initialisation

Initialisation of the various software components used with the OTA Upgrade cluster (see Section 20.4) must be performed in a particular order in the application code. The initialisation could be incorporated in a function **APP_vInitialise()**, as is the case in the NXP *ZigBee PRO Application Template (JN-AN-1123)*.

Initialisation must be performed in the following order:

1. The JenOS RTOS must first be started using the function **OS_vStart()**.
2. The PDM module must next be initialised using the function **PDM_vInit()**.
3. If using a JN5148 device, overlays must now be initialised using the function **OVLY_bInit()** - this function is described in the *JenOS User Guide (JN-UG-3075)*. The initialisation data provided to this function includes pointers to the callback functions for getting and releasing a mutex (see Section 20.7.6). Also note that:
    - The variable *u16ImageStartSector*, which is defined in the linker command file, needs to be declared as a **uint16** 'extern'. This variable contains the number of the start sector in Flash memory of the space for the first image (note that it 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).
    - The *u32ImageOffset* parameter of the intialisation data for the function **OVLY_bInit()** must be set to *u16ImageStartSector* x 0x8000.
4. The persistent data record(s) should then be initialised using the function **PDM_eLoadRecord()**.
5. The ZigBee PRO stack must now be started by first calling the function **ZPS_vSetOverrideLocalMacAddress()** to over-ride the existing MAC address (JN516x only), followed by **ZPS_eAplAfInit()** to initialise the Application Framework and then **ZPS_eAplZdoStartStack()** to start the stack.
6. The initialisation function for the relevant ZigBee application profile can now be called. An OTA Upgrade cluster instance should then be created using **eOTA_Create()** (this call is not needed for Smart Energy), followed by a call to **eOTA_UpdateClientAttributes()** or **eOTA_RestoreClientData()** on a client to initialise the cluster attributes.
7. The Flash programming of the OTA Upgrade cluster must now be initialised using the function **vOTA_FlashInit()**. If an unsupported/custom Flash memory device is used, callback functions must be provided to perform read, write, erase and initialisation operations, otherwise standard NXP callback functions are used - see function description on page 385.
8. The required device endpoint(s) from the relevant application profile can now be registered (e.g. an IPD from the Smart Energy profile).
9. 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.
10. On the server, a set of client devices can be defined for which OTA upgrades are authorised - 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()**.

11. 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 PRO Stack User Guide (JN-UG-3048)*. 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 20.6.

20.6 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:

   a) 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.

   a) A signature can be generated and attached to the image, as described in Section 20.7.9.

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:

   - **Co-ordinator or Router client:** The server can notify the Co-ordinator 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:

   - **Co-ordinator 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 20.12.

   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 may generate the callback event E_CLD_OTA_INTERNAL_COMMAND_OTA_START_IMAGE_VERIFICATION_IN_LOW_PRIORITY, depending whether or not the image was signed. If signed, the application should validate the image using the steps described in Section 20.7.9 before transmitting an Upgrade End Request to the server (i.e. by calling **eOTA_HandleImageVerification()**):

   This Upgrade End Request may report success or an invalid image (provided that the return code from **eOTA_VerifyImage()** is passed into the **eOTA_HandleImageVerification()** function). 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 will proceed as follows (although the details will be manufacturer-specific):

   a) A reboot of the JN51xx device will be initiated causing the default bootloader to run.

   b) The running bootloader will find the (only) valid application image in external Flash memory and load it into RAM or internal Flash memory (JN5148-Z01 or JN516x, respectively).

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 will then be 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.

20.7 Ancillary Features and Resources for OTA Upgrade

As indicated in Section 20.4, in order to implement OTA upgrades, a number of other software features and resources are available. These are described in the sub-sections below.

20.7.1 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 prioritised 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 6.

'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 20.2) 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. It is a software timer that is set up and controlled using the JenOS RTOS - for details, refer to the *JenOS User Guide (JN-UG-3075)*.

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 using the JenOS function **OS_eStartSWTimer()** and will expire after the specified interval has passed. This expiry 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 using the JenOS function **OS_eStopSWTimer()**.
- Otherwise, the application must start the next timed interval (until the next request) by calling the JenOS function **OS_eContinueSWTimer()**.

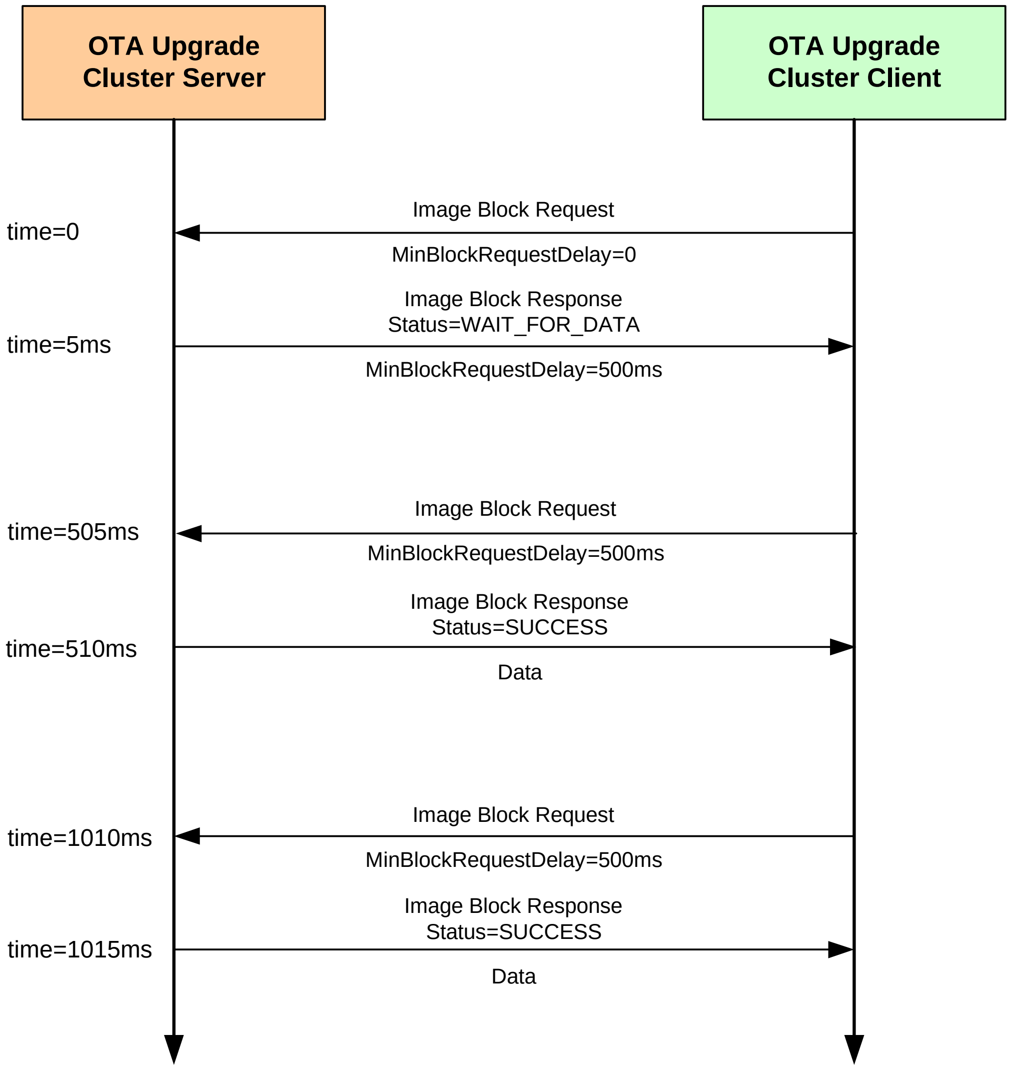

OTA Upgrade Cluster Server
OTA Upgrade Cluster Client
time=0
Image Block Request
MinBlockRequestDelay=0
time=5ms
Image Block Response Status=WAIT_FOR_DATA
MinBlockRequestDelay=500ms
time=505ms
Image Block Request
MinBlockRequestDelay=500ms
time=510ms
Image Block Response Status=SUCCESS
Data
time=1010ms
Image Block Request
MinBlockRequestDelay=500ms
time=1015ms
Image Block Response Status=SUCCESS
Data

Figure 6: Example 'Rate Limiting' Exchange

20.7.2 Device-Specific File Downloads

An OTA Ugrade client can request a file (from the server) which 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_ClientQuerySpecificFileRequest()**.	
2		On arrival at the server, the Query Specific File Request triggers the event E_CLD_OTA_COMMAND_QUERY_SPECIFIC _FILE_REQUEST.
3		Server automatically replies to the request with a Query Specific File Response - the application can also send a response using **eOTA_ServerQuerySpecificFileResponse()**.
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).	

	On Client	On Server
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_FILE_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.

20.7.3 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 20.6, 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 20.12.

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). It is a software timer that is set up and controlled using the JenOS RTOS - for details, refer to the *JenOS User Guide (JN-UG-3075)*.

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 using the JenOS function **OS_eStartSWTimer()** and will expire after the specified interval has passed. This expiry 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 using the JenOS function **OS_eStopSWTimer()**.
- Otherwise, the application must start the next timed interval (until the next response) by calling the JenOS function **OS_eContinueSWTimer()**.

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 20.12):

- 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 20.6 for standard OTA downloads.

20.7.4 Use of Overlays (JN5148 only)

Overlays (paging) should normally be enabled for applications that participate in OTA upgrades. In order to use overlays, the application must include the overlay header file **ovly.h** and overlays must be configured as described in the *JenOS User Guide (JN-UG-3075)*. Care must be taken in the application to allocate enough memory space for the overlay initialisation structure of the type **OVLY_tsInitData**, which is provided through the overlay initialisation function **OVLY_bInit()**.

The overlay mechanism requires a mutex for accessing Flash memory (see Section 20.7.6). Callback functions for implementing this mutex are provided as part of the overlay initialisation data.

20.7.5 Persistent Data Management

The OTA Upgrade cluster on a client requires context data to be preserved in Flash memory to facilitate a recovery of the OTA Upgrade status following a device reboot. The JenOS Persistent Data Manager (PDM) module should be used to perform this data saving and recovery. The PDM module is implemented as described in the *JenOS User Guide (JN-UG-3075)*.

Persistent data should normally be stored in the final sector of Flash memory (e.g. Sector 7 of an 8-sector device) or EEPROM (JN5148 or JN516x, respectively). Thus, when the PDM module is initialised, this sector should be specified (just this one sector should be managed by the PDM 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 to Flash memory. A user-defined callback function can then be invoked to perform the data storage using functions of the PDM module.

The OTA Upgrade cluster is implemented for an individual application/endpoint. Therefore, the PDM 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;
PUBLIC PDM_tsRecordDescriptor s_OTAPDDesc;
```

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()**.

20.7.6 Mutex for Flash Memory Access

The Flash memory device on a node is accessed from the JN51xx device via the SPI bus. Flash memory needs to be accessed by the OTA Upgrade cluster, the overlay mechanism and the Persistent Data Manager (PDM). Each access should be allowed to complete before allowing the next access to start and, therefore, should be protected by a mutex.

A JenOS mutex can be used, as described in the *JenOS User Guide (JN-UG-3075)*. Callback functions should be defined which allow the OTA Upgrade cluster to get and release a Flash memory mutex, as illustrated below.

```
void vGrabLock(void)
{
     OS_eEnterCriticalSection(mutexFLASH);
}

void vReleaseLock(void)
{
     OS_eExitCriticalSection(mutexFLASH);
}
```

These callback functions are provided as part of the overlay initialisation (see Section 20.7.4) and are invoked when the following events are generated for the application:

E_CLD_OTA_INTERNAL_COMMAND_LOCK_FLASH_MUTEX
E_CLD_OTA_INTERNAL_COMMAND_FREE_FLASH_MUTEX

Note: The above user-defined callback functions to get and release a mutex must be designed such that OTA Upgrade and overlays are in the same mutex group as the PDM module. If the mutex is not properly implemented, unpredictable behaviour may result.

20.7.7 Flash Memory Organisation

Flash memory should be organised such that the application images are stored from Sector 0 and persistent data is stored in the final sector.

Thus, for a Flash memory device with 8 sectors:

- Sectors 0-6 are available for the storage of application images
- Sector 7 is used for persistent data storage (a JN516x device may use internal EEPROM instead, in which case sector 7 will be available for application storage)

Storage of the above software is described further below.

Application Images

As part of application initialisation (see Section 20.5), 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).
- Number of images for the endpoint (the maximum number of images per endpoint is specified in the **zcl_options.h** file - see Section 20.12)
- 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 PDM module (see Section 20.7.5) and the sector used is specified as part of the PDM initialisation through **PDM_vInit()** - the final sector of external Flash memory should be specified (if not using the internal EEPROM on the JN516x device).

20.7.8 Security (Smart Energy only)

In the case of a ZigBee PRO Smart Energy network, security must be applied to network communications by means of the Key Establishment cluster from the Smart Energy profile. Thus, the Key Establishment cluster must be enabled for use with the OTA Upgrade cluster. For details of Smart Energy security and the Key Establishment cluster, refer to the *ZigBee PRO Smart Energy API User Guide (JN-UG-3059)*.

In order to set up security between the server and a particular client, a security certificate and associated private key must be obtained from a Certificate Authority (CA), such as Certicom - the certificate also contains the CA’s public key. A pre-configured link key is also required for the client. These keys must be set for the different application images as described below. In all cases, the certificate, public key and private key can be registered with the Key Establishment cluster using the following function call:

```
eSE_KECLoadKeys(LOCAL_EP, au8CAPublicKey, au8Certificate, au8PrivateKey);
```

Server Image

The certificate, private key and link key for the server must be set in the server application in the same way as described for the first client image (see First Client Image below).

First Client Image

In the first ever image for the client, the certificate, private key and link key can be set near the start of the application code as illustrated in the following example (values within the certificate/keys are just for illustration):

```
PUBLIC uint8 au8Certificate[48] __attribute__ ((section
(".ro_se_cert"))) = {0x03, 0x07, 0x17, 0xa9, 0xc0, 0xdc, 0x57,
 0x18, 0xfd, 0xc4, 0xf7, 0xa9, 0x92, 0x83, 0xe0, 0x8f, 0x1c,
 0xea, 0xfa, 0x65, 0x30, 0xcf, 0x00, 0x00, 0x00, 0x00, 0x01,
 0x00, 0x00, 0x00, 0x54, 0x45, 0x53, 0x54, 0x53, 0x45, 0x43,
 0x41, 0x01, 0x09, 0x10, 0x83, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00};

PUBLIC uint8 au8PrivateKey[21] __attribute__ ((section
(".ro_se_pvKey"))) = {0x01, 0xa5, 0x37, 0x20, 0xa5, 0x1f, 0x3a,
0xc6, 0x86, 0x9e, 0x2e, 0x8a, 0x15, 0x3f, 0xf7, 0x75, 0xc4, 0xa3,
0xf5, 0x43, 0x4c};

PUBLIC uint8 s_au8LnkKeyArray[16] __attribute__ ((section
(".ro_se_lnkKey"))) = {0xFF, 0x11, 0x22, 0x33, 0x44, 0x55, 0x66,
0x77, 0x88, 0x99, 0xAA, 0xBB, 0xCC, 0xDD, 0xEE, 0x00};
```

Alternatively, the values within the certificate/keys can all be set to 0xFF (see example in Upgrade Client Image below) and the actual values can be later set directly in Flash memory using the Jennic Encryption Tool (JET), described in the *JET User Guide (JN-UG-3081)*.

Upgrade Client Image

In an upgrade image for the client, the values within the certificate, private key and link key must all be set to 0xFF in the application code, as illustrated below.

```
PUBLIC uint8 au8Certificate[48] __attribute__ ((section
(".ro_se_cert"))) = {0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff,
0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff,
0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff,
0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff,
0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff};

PUBLIC uint8 au8PrivateKey[21] __attribute__ ((section
(".ro_se_pvKey"))) = {0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff,
0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff,
0xff, 0xff, 0xff};

PUBLIC uint8 s_au8LnkKeyArray[16] __attribute__ ((section
(".ro_se_lnkKey"))) = {0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff,
0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff};
```

The 0xFF values are then replaced in Flash memory by the values set for the first client image (see First Client Image above) as part of the upgrade process.

20.7.9 Signatures (Optional)

Signatures can be optionally used in the download of application images from server to client in order to verify that an image comes from a valid server and that the image has not been corrupted during download. A signature is generated and appended to the image to be downloaded with the help of JET (JN-SW-4052). On completion of the download, the client checks the attached signature - if the signature fails verification then the image is discarded.

Use of signatures requires that the following information (in addition to the image data) is available on the client:

- IEEE/MAC address of the signer
- Public key of Certificate Authority (CA)
- Public key of signer

The public keys are extracted from a security certificate which is obtained from a Certificate Authority, such as Certicom. In addition, a private key is needed by the signer - this key is also supplied with the certificate by the CA.

The server's security certificate is also appended to the downloaded image, providing the client with the necessary information to validate the attached signature.

The implementation of signatures on the client is described below. Much more detailed accounts of the signature generation and validation processes can be found in the *ZigBee Over-the-Air Upgrading Cluster Specification (095264)* from the ZigBee Alliance.

Signature Validation on Client

A client must be configured to validate signed images from the server by including the following line in the **zcl_options.h** file:

```
#define OTA_ACCEPT_ONLY_SIGNED_IMAGES
```

If this option is set, only images with signatures will be accepted by the client. If the option is not set, images without signatures will be accepted (but no signature validation will be implemented).

Once the final block of image data has been received, the client will generate the callback event E_CLD_OTA_INTERNAL_COMMAND_OTA_START_IMAGE_VERIFICATION_IN_LOW_PRIORITY. On receiving this event, the application should call the function **eOTA_VerifyImage()** from a low-priority task, the return code of which must be passed into the **eOTA_HandleImageVerification()** function. For an illustration of this low-priority task, refer to the code fragment in Appendix F.1.

The validation of a signature in a received image is outlined below (if any check fails, the image is discarded):

1. The signer’s IEEE/MAC address is extracted from the image and the event E_CLD_OTA_INTERNAL_COMMAND_VERIFY_SIGNER_ADDRESS is generated to prompt the application to verify this address, which is included in the event. The application must check this address against the client’s list of approved signers and then set the status field of the event to one of the following (also refer to the code fragment below):
   - E_ZCL_SUCCESS if the signer’s address is in the list
   - E_ZCL_FAIL if the signer’s address is not in the list (in this case, the image will be discarded)
2. Provided that signer’s address has been checked as valid by the application, the signer’s security certificate is extracted from the image and the signer’s IEEE/MAC address within the certificate is checked against the IEEE/MAC address previously extracted from the image (in Step 1).
3. The CA public key within the certificate is used to check that the Certificate Authority is known to the client.
4. A checksum is calculated from the received image. This value is then used together with the CA public key and signer’s public key (from the certificate) to generate a signature.
5. The locally generated signature is compared with the signature appended to the image - if they match, the image is valid.

20.8 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 20.11.2). The events are listed in the table below, which also indicates on which side of the cluster (server or client) the events can occur:

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
	E_CLD_OTA_INTERNAL_COMMAND_SPECIFIC_FILE_NO_UPGRADE_END_RESPONSE
	E_CLD_OTA_COMMAND_QUERY_NEXT_IMAGE_RESPONSE_ERROR

Table 15: OTA Upgrade Events

Cluster Side(s)	Event
	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
Both	E_CLD_OTA_INTERNAL_COMMAND_LOCK_FLASH_MUTEX
	E_CLD_OTA_INTERNAL_COMMAND_FREE_FLASH_MUTEX

Table 15: OTA Upgrade Events

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.

20.8.1 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.

- **E_CLD_OTA_INTERNAL_COMMAND_CO_PROCESSOR_IMAGE_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 and the server finds that the required image is stored in the co-processor's external storage device. The JN51xx application can then fetch the required image block from the co-processor and send it in an Image Block Response to the client (whose address and endpoint details are contained in the event).

20.8.2 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_CO_PROCESSOR_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 the co-processor image. Following this event, the JN51xx application can store the block in the appropriate place (attached Flash memory or co-processor's storage device). The client can also 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_INTERNAL_COMMAND_CO_PROCESSOR_DL_ABORT**

  This event is generated on the client to notify the application that the download of the co-processor image from the server has been aborted.

- **E_CLD_OTA_INTERNAL_COMMAND_CO_PROCESSOR_IMAGE_DL_ COMPLETE**

  This event is generated on the client to notify the application that the download of the co-processor image from the server has completed (all blocks have been received). Following this event, the JN51xx application should verify the image and call **eOTA_CoProcessorUpgradeEndRequest()** to send an Upgrade End Request to the server.

- **E_CLD_OTA_INTERNAL_COMMAND_CO_PROCESSOR_SWITCH_TO_NEW_IMAGE**

  This event is generated on the client to notify the application that the upgrade time for a previously downloaded co-processor image has been reached. This event occurs after receiving the Upgrade End Response which contains the upgrade time. Following this event, the JN51xx application should instruct the co-processor to update its own running application image.

- **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_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 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_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 in order to allow it to send a Query Next Image command for its own upgrade image, using **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 JN51xx client image from a low priority task. Once the low priority task is running, the application should call **eOTA_VerifyImage()** to begin 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 serialisation data from the active image to the new upgrade image fails. This process takes place after image and signature validation (if applicable) is completed successfully.

20.8.3 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 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 Flash memory (via the SPI bus).

20.9 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 20.9.1
- Server functions - see Section 20.9.2
- Client functions - see Section 20.9.3

Note: When referring to the storage of OTA upgrade images in Flash memory, this is a Flash memory device which is external to the JN51xx device (i.e. not the internal Flash memory, in the case of JN516x).

20.9.1 General Functions

The following OTA Upgrade cluster functions can be used on the cluster server and the cluster client:

Function	Page
eOTA_Create	384
vOTA_FlashInit	385
eOTA_AllocateEndpointOTASpace	386
eOTA_VerifyImage	388
vOTA_GenerateHash	389
eOTA_GetCurrentOtaHeader	390

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 should only be called when the OTA Upgrade cluster will be used in a non-SE application. In this case, it must be the first OTA function called in the application, and must be called after the stack has been started and after the application profile has been initialised. In the case of Smart Energy, this function is called internally by the function **eSE_Initialise()**, in which case there is no need for the application to call it explicitly.

Parameters

psClusterInstance	Pointer to structure containing information about the cluster instance to be created (see Section 23.1.15)
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 23.1.2) - 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. ESP)
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 20.10.2)

Returns

E_ZCL_SUCCESS

E_ZCL_FAIL

vOTA_FlashInit

```
void vOTA_FlashInit(void *pvFlashTable,
                    tsNvmDefs *psNvmStruct);
```

Description

This function initialises 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 initialisation 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 initialisation 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 20.10.4

Returns

None

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 will be 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.

Note that:

- For the JN5148-001 device, the first sector (0) is reserved for the second-stage bootloader used by the OTA Upgrade cluster
- For both JN5148 variants, the final sector is reserved for any persistent data storage (so, Sector 7 in the case of an 8-sector Flash memory device)

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 will be 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

eOTA_VerifyImage

```
teZCL_Status  eOTA_VerifyImage(uint8 u8Endpoint,
                               bool bIsServer,
                               uint8 u8ImageLocation,
                               bool bHeaderPresent);
```

Description

This function can be used to verify a signed image in Flash memory. The function generates a signature from the image and compares it with the signature appended to the image. If the signatures do not match, the image should be discarded.

Parameters

u8Endpoint	Number of endpoint corresponding to application
bIsServer	Side of cluster implemented on this device: TRUE - Server FALSE - Client
u8ImageLocation	Number of sector where image starts in Flash memory
bHeaderPresent	Presence of image header: TRUE - Present FALSE - Absent

Returns

E_ZCL_SUCCESS

E_ZCL_FAIL

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 23.1.1)
psCustomData	Pointer to data structure connected with event associated with the checksum (see Section 20.10.2)
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

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 20.10.1)

Returns

E_ZCL_SUCCESS

E_ZCL_ERR_EP_RANGE

E_ZCL_ERR_CLUSTER_NOT_FOUND

E_ZCL_ERR_PARAMETER_NULL

20.9.2 Server Functions

The following OTA Upgrade cluster functions can be used on the cluster server only:

Function	Page
eOTA_SetServerAuthorisation	392
eOTA_SetServerParams	393
eOTA_GetServerData	394
eOTA_EraseFlashSectorsForNewImage	395
eOTA_FlashWriteNewImageBlock	396
eOTA_NewImageLoaded	397
eOTA_ServerImageNotify	398
eOTA_ServerQueryNextImageResponse	399
eOTA_ServerImageBlockResponse	400
eOTA_SetWaitForDataParams	402
eOTA_ServerUpgradeEndResponse	403
eOTA_ServerSwitchToNewImage	405
eOTA_InvalidateStoredImage	406
eOTA_ServerQuerySpecificFileResponse	407

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 will be authorised to download application images. The function allows all clients to be authorised or a list of selected authorised 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 authorised clients will be used or all clients will be authorised - one of: E_CLD_OTA_STATE_USE_LIST E_CLD_OTA_STATE_ALLOW_ALL
pu64WhiteList	Pointer to list of IEEE/MAC addresses of authorised clients (ignored if all clients are authorised through *eState* parameter)
u8Size	Number of clients in list (ignored if all clients are authorised through *eState* parameter)

Returns

E_ZCL_SUCCESS

E_ZCL_FAIL

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 20.10.22. 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 will be 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 JN51xx external Flash memory, this index will take a value in the range 0 to (OTA_MAX_IMAGES_PER_ENDPOINT - 1). In the case of a dual-processor OTA server node, refer to Appendix D.4.

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 20.10.22)

Returns

E_ZCL_SUCCESS

E_ZCL_FAIL

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 20.10.22. 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 JN51xx external Flash memory, this index will take a value in the range 0 to (OTA_MAX_IMAGES_PER_ENDPOINT - 1). In the case of a dual-processor OTA server node, refer to Appendix D.4.

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 20.10.22)

Returns

E_ZCL_SUCCESS
E_ZCL_FAIL

eOTA_EraseFlashSectorsForNewImage

```
teZCL_Status eOTA_EraseFlashSectorsForNewImage(
                                    uint8 u8Endpoint,
                                    uint8 u8ImageIndex);
```

Description

This function can be used to erase certain sectors of the Flash memory attached to the JN51xx 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

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 Flash memory attached to the JN51xx device 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 (this image may be destined for the JN51xx device or a co-processor in the OTA client node)

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).

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

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 external Flash memory of the JN51xx device or in the external storage device of a co-processor (if any) within the server node:

- For an image stored in JN51xx external 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.

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 storage FALSE - Stored in JN51xx 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

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 will be sent
u8DestinationEndpoint	Number of endpoint (on client) to which the message will be 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 23.1.4)
psImageNotifyCommand	Pointer to structure containing payload for message (see Section 20.10.5)

Returns

E_ZCL_SUCCESS

E_ZCL_FAIL

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 will be set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

Parameters

u8SourceEndpoint	Number of endpoint (on server) from which the response will be sent
u8DestinationEndpoint	Number of endpoint (on client) to which the response will be sent
psDestinationAddress	Pointer to structure containing the address of the target client for the response (see Section 23.1.4)
psQueryImageResponsePayload	Pointer to structure containing payload for response (see Section 20.10.7)
u8TransactionSequenceNumber	Pointer to a location to store the Transaction Sequence Number (TSN) of the request

Returns

E_ZCL_SUCCESS

E_ZCL_FAIL

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 20.12).

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 will be set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

Parameters

u8SourceEndpoint	Number of endpoint (on server) from which the response will be sent
u8DestinationEndpoint	Number of endpoint (on client) to which the response will be sent
psDestinationAddress	Pointer to structure containing the address of the target client for the response (see Section 23.1.4)
psImageBlockResponsePayload	Pointer to structure containing payload for response (see Section 20.10.10)
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

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 20.7.1.

Parameters

u8Endpoint	Number of endpoint (on server) from which the response will be sent
u16ClientAddress	Network address of client device to which the response will be sent
sWaitForDataParams	Pointer to structure containing ‘Wait for Data’ parameter values for Image Block Response payload (see Section 20.10.14)

Returns

E_ZCL_SUCCESS

E_ZCL_FAIL

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 will be set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

Parameters

u8SourceEndpoint	Number of endpoint (on server) from which the response will be sent
u8DestinationEndpoint	Number of endpoint (on client) to which the response will be sent
psDestinationAddress	Pointer to structure containing the address of the target client for the response (see Section 23.1.4)
psUpgradeResponsePayload	Pointer to structure containing payload for response (see Section 20.10.12)
u8TransactionSequenceNumber	Pointer to a location to store the Transaction Sequence Number (TSN) of the request

Returns

E_ZCL_SUCCESS

E_ZCL_FAIL

eOTA_ServerSwitchToNewImage

```
teZCL_Status eOTA_ServerSwitchToNewImage(
                                uint8 u8Endpoint,
                                uint8 u8ImageIndex);
```

Description

This function can be used to force a reset of the JN51xx 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 JN51xx 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

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 attached to the JN51xx device in the OTA server node. 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

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 will be set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

Parameters

u8SourceEndpoint	Number of endpoint (on server) from which the response will be sent
u8DestinationEndpoint	Number of endpoint (on client) to which the response will be 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 20.10.19)
u8TransactionSequenceNumber	Pointer to a location to store the Transaction Sequence Number (TSN) of the request

Returns

E_ZCL_SUCCESS

E_ZCL_FAIL

20.9.3 Client Functions

The following OTA Upgrade cluster functions can be used on the cluster client only:

Function	**Page**
eOTA_SetServerAddress	409
eOTA_ClientQueryNextImageRequest	410
eOTA_ClientImageBlockRequest	411
eOTA_ClientImagePageRequest	412
eOTA_ClientUpgradeEndRequest	413
eOTA_HandleImageVerification	415
eOTA_ClientSwitchToNewImage	416
eOTA_UpdateCoProcessorOTAHeader	417
eOTA_CoProcessorUpgradeEndRequest	418
eOTA_UpdateClientAttributes	419
eOTA_RestoreClientData	420
vOTA_SetImageValidityFlag	421
eOTA_ClientQuerySpecificFileRequest	422
eOTA_SpecificFileUpgradeEndRequest	423

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 PRO Stack User Guide (JN-UG-3048)*. 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

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 will be sent
u8DestinationEndpoint	Number of endpoint (on server) to which the request will be sent
psDestinationAddress	Pointer to structure containing the address of the target server (see Section 23.1.4)
psQueryImageRequest	Pointer to structure containing payload for request (see Section 20.10.6)

Returns

E_ZCL_SUCCESS

E_ZCL_FAIL

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 will be sent
u8DestinationEndpoint	Number of endpoint (on server) to which the request will be sent
psDestinationAddress	Pointer to structure containing the address of the target server (see Section 23.1.4)
psOtaBlockRequest	Pointer to structure containing payload for request (see Section 20.10.8)

Returns

E_ZCL_SUCCESS

E_ZCL_FAIL

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 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 20.12).

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 20.7.3.

Parameters

u8SourceEndpoint	Number of endpoint (on client) from which the request will be sent
u8DestinationEndpoint	Number of endpoint (on server) to which the request will be sent
psDestinationAddress	Pointer to structure containing the address of the target server (see Section 23.1.4)
psOtaPageRequest	Pointer to structure containing payload for request (see Section 20.10.9)

Returns

E_ZCL_SUCCESS

E_ZCL_FAIL

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 will be sent
u8DestinationEndpoint	Number of endpoint (on server) to which the request will be sent
psDestinationAddress	Pointer to structure containing the address of the target server (see Section 23.1.4)

psUpgradeEndRequestPayload	Pointer to structure containing payload for request, including reported status (see Section 20.10.11)

Returns

E_ZCL_SUCCESS

E_ZCL_FAIL

eOTA_HandleImageVerification

```
teZCL_Status eOTA_HandleImageVerification(
                        uint8 u8SourceEndPointId,
                        uint8 u8DstEndpoint,
                        teZCL_Status eImageVerificationStatus);
```

Description

This function should be called after calling **eOTA_VerifyImage()**, the result of which should be passed into this function using the *eImageVerificationStatus* parameter. 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 will be sent
eImageVerificationStatus	Returned status code from **eOTA_VerifyImage()**

Returns

E_ZCL_FAIL
E_ZCL_SUCCESS

eOTA_ClientSwitchToNewImage

```
teZCL_Status eOTA_ClientSwitchToNewImage(
                        uint8 u8SourceEndPointId);
```

Description

This function is used to switch a JN51xx device to a new client image when a co-processor upgrade is a dependent, i.e. all upgrade images are required to complete at the same time. This function should be called from the callback event E_CLD_OTA_INTERNAL_COMMAND_CO_PROCESSOR_SWITCH_TO_NEW_IMAGE.

Parameters

u8SourceEndPointId	Identifier of endpoint on which the cluster client operates

Returns

E_ZCL_ERR_PARAMETER_NULL

E_ZCL_SUCCESS

eOTA_UpdateCoProcessorOTAHeader

```
teZCL_Status eOTA_UpdateCoProcessorOTAHeader(
            tsOTA_CoProcessorOTAHeader
                          *psOTA_CoProcessorOTAHeader
            bool_t bIsCoProcessorImageUpgradeDependent);
```

Description

This function can be used to register the OTA header information of one or more co-processor upgrade image(s) with the OTA Upgrade cluster client before the client requests a download of the image(s) from the OTA server node. The function also specifies whether or not the co-processor image(s) are dependent on the client image that is also being upgraded.

Parameters

psOTA_CoProcessorOTAHeader	Pointer to the OTA header of a co-processor upgrade image.
bIsCoProcessorImageUpgradeDependent	Indicates whether the co-processor upgrade image is dependant on the client image that is also being upgraded: TRUE - Image upgrade is dependent on other upgrade images FALSE - Image upgrade is independent

Returns

E_ZCL_ERR_PARAMETER_NULL

E_ZCL_SUCCESS

eOTA_CoProcessorUpgradeEndRequest

```
teZCL_Status eOTA_CoProcessorUpgradeEndRequest(
                        uint8 u8SourceEndPointId,
                        uint8 u8Status);
```

Description

This function can be used during the download of a co-processor upgrade image 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 application 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 response triggers an E_CLD_OTA_COMMAND_UPGRADE_END_RESPONSE event.

Note: Although the OTA Upgrade cluster client normally sends an Upgrade End Request to the server on completion of a download, this is not the case for a co-processor image and so it is necessary for the application to call this function.

Parameters

u8SourceEndPointId	Number of endpoint (on client) on which cluster operates
u8Status	Status of download and verification, one of: OTA_STATUS_SUCCESS OTA_STATUS_INVALID_IMAGE OTA_REQUIRE_MORE_IMAGE OTA_STATUS_ABORT

Returns

E_ZCL_SUCCESS

E_ZCL_FAIL

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 initialisation after the cluster instance has been created using **eOTA_Create()** (or, in the case of Smart Energy, after **eSE_Initialise()** has been called).

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

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 JenOS Persistent Data Manager) 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 20.10.13)
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

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 20.10.2)
bSet	Flag state to be set: TRUE - Reset FALSE - No reset
psEndPointDefinition	Pointer to endpoint definition (see Section 23.1.1)

Returns

None

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 will be sent
u8DestinationEndpoint	Number of endpoint (on server) to which the request will be 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

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 will be 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

20.10 Structures

20.10.1 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.

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 wild card 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 wild card value of 0xFFFF represents any file type):

Value	File Type
0x0000 – 0xFFBF	Manufacturer-specific
0xFFC0	Security credential
0xFFC1	Configuration
0xFFC2	Log
0xFFC3 – 0xFFFE	Reserved
0xFFFF	Wild card

- `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 such as signature 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

20.10.2 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 20.10.21.

20.10.3 tsOTA_HwFncTable

The following structure contains pointers to callback functions to be used by the OTA Upgrade cluster to perform initialisation, 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 initialisation 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)

20.10.4 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 initialisation, erase, write and read operations on the Flash memory device (see Section 20.10.3) - 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_ST_M25P10_A (ST M25P10A)
    - E_FL_CHIP_ST_M25P40_A (ST M25P40)
    - E_FL_CHIP_SST_25VF010 (SST 25VF010)
    - E_FL_CHIP_ATMEL_AT25F512 (Atmel AT25F512)
    - E_FL_CHIP_CUSTOM (custom device)
    - E_FL_CHIP_AUTO (auto-detection)

20.10.5 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 20.11.4)
- `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 20.6

20.10.6 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)

20.10.7 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.

20.10.8 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)

20.10.9 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 will be 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)

20.10.10 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 20.10.14) - 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 20.10.13) - this data is only provided in the case of the status OTA_STATUS_SUCCESS

20.10.11 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)

20.10.12 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)

20.10.13 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 20.10.10):

```
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)

20.10.14 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 20.10.10). 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

20.10.15 tsOTA_WaitForDataParams

The following structure is used in the `tsOTA_CallBackMessage` structure (see Section 20.10.21) 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 20.10.14)

20.10.16 tsOTA_PageReqServerParams

The following structure is used in the `tsOTA_CallBackMessage` structure (see Section 20.10.21) 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 20.10.9)
- `sReceiveEventAddress` contains the address of the OTA Upgrade client that made the page request

20.10.17 tsOTA_PersistedData

The following structure contains the persisted data that is stored in Flash memory using the JenOS PDM module:

```
typedef struct
{
    tsCLD_AS_Ota sAttributes;
    tsZCL_Address sDestinationAddress;
    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];
#if JENNIC_CHIP_FAMILY == JN514x
    uint8 u8CurrentSigningCertificate [OTA_SIGNING_CERT_SIZE];
    uint8 u8CurrentSignature[OTA_SIGNITURE_SIZE];
#endif
    uint8 au8Header[OTA_MAX_HEADER_SIZE];
    uint8 u8Retry;
    uint8 u8RequestTransSeqNo;
    uint8 u8DstEndpoint;
    bool_t bIsCoProcessorImage;
    bool_t bIsSpecificFile;
    bool_t bIsNullImage;
    uint8  u8CoProcessorOTAHeaderIndex;
#if JENNIC_CHIP_FAMILY == JN514x
    AESSW_Block_u uClientHash;
    AESSW_Block_u uClientBufToHash;
    uint8 u8ClientRemainingLengthToHash;
#endif
    uint32 u32CoProcessorImageSize;
    uint32 u32SpecificFileSize;
#ifdef OTA_PAGE_REQUEST_SUPPORT
    tsOTA_PageReqParams sPageReqParams;
#endif
#if (OTA_MAX_CO_PROCESSOR_IMAGES != 0)
    uint8 u8NumOfDownloadableImages;
#endif
}tsOTA_PersistedData;
```

The fields are for internal use and no knowledge of them is required.

20.10.18 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

20.10.19 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 20.10.18)
- `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 20.10.18)
- `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 20.10.18)
- `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.

20.10.20 tsOTA_SignerMacVerify

The following structure contains the data for an event of the type E_CLD_OTA_INTERNAL_COMMAND_VERIFY_SIGNER_ADDRESS.

```
typedef struct
{
    uint64 u64SignerMac;
    teZCL_Status eMacVerifyStatus;
}tsOTA_SignerMacVerify;
```

where:

- `u64SignerMac` is the IEEE/MAC address of the device which signed the OTA upgrade image
- `eMacVerifyStatus` is a status field which should be updated to E_ZCL_SUCCESS or E_ZCL_FAIL by the application after verification of signer address, to indicate whether the upgrade image has come from a trusted source

20.10.21 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 20.11.2)
- `sPersistedData` is the structure (see Section 20.10.17) which contains the persisted data that is stored in Flash memory using the JenOS PDM 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 20.10.14)
- `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 JN51xx external 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)

20.10.22 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 20.10.10 and Section 20.10.14)
  - As an ‘upgrade time’, the value will be included in an Upgrade End Response (see Section 20.10.12)
- `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 20.6
- `u8DataSize` is the length, in bytes, of the data block pointed to by `pu8Data`

20.10.23 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 Section 20.2.

20.10.24 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

20.10.25 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

20.11 Enumerations

20.11.1 teOTA_Cluster

The following enumerations represent the OTA Upgrade cluster attributes:

```
typedef enum PACK
{
    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.

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

Table 16: OTA Upgrade Cluster Attributes

The above attributes are described in Section 20.2.

20.11.2 teOTA_UpgradeClusterEvents

The following enumerations represent the OTA Upgrade cluster events:

```
typedef enum PACK
{
    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.

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
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

Table 17: OTA Upgrade Cluster Events

Enumeration	Event Description
E_CLD_OTA_INTERNAL_COMMAND_RESET_TO_ UPGRADE	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 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 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 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

Table 17: OTA Upgrade Cluster Events

Enumeration	Event Description
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 versoin, 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_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()**.

Table 17: OTA Upgrade Cluster Events

Enumeration	Event Description
E_CLD_OTA_INTERNAL_COMMAND_OTA_START_ IMAGE_VERIFICATION_IN_LOW_PRIORITY	This event is generated to prompt the application to verify the downloaded JN51xx 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 serialisation data from the active image to the new upgrade image fails. This process takes place after image and signature validation (if applicable) are completed successfully.

Table 17: OTA Upgrade Cluster Events

The above events are described in more detail in Section 20.8.

20.11.3 eOTA_AuthorisationState

The following enumerations represent the authorisation options concerning which clients are allowed to obtain upgrade images from the server:

```
typedef enum PACK
{
    E_CLD_OTA_STATE_ALLOW_ALL,
    E_CLD_OTA_STATE_USE_LIST
}eOTA_AuthorisationState;
```

The above enumerations are described in the table below.

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

Table 18: Client Authorisation Options

20.11.4 teOTA_ImageNotifyPayloadType

The following enumerations represent the payload options for an Image Notify message issued by the server:

```
typedef enum PACK
{
    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.

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 Jitter’ in payload
E_CLD_OTA_ITYPE_MDID_FVERSION_JITTER	Include ‘Image Type’, ‘Manufacturer Code’, ‘File Version’ and ‘Query Jitter’ in payload

Table 19: Image Notify Payload Options

20.12 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
```

Number of Images

The maximum number of images that can be stored in the external Flash memory of the JN51xx 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
```

The smallest value that should be used for a JN5148 client or server is 2 in order to cover the currently active image and an upgrade image. The smallest value that should be used for a JN516x client or server is 1, as the active image is stored in internal Flash memory.

In the case of a dual-processor client or server node, the maximum number of images that can be stored in the co-processor's external storage device must be specified as follows, where in this example the maximum is one image:

```
#define OTA_MAX_CO_PROCESSOR_IMAGES             1
```

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 PRO Stack User Guide (JN-UG-3048)*.

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 Serialisation Data

To maintain custom serialisation 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 acks will be 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)

Signed Images (Client only)

If the image to be accepted is signed by the server, the following needs to be defined on the client in order for the signature to be verified:

```
#define OTA_ACCEPT_ONLY_SIGNED_IMAGES
```

Device Address Copying

On a JN51xx 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

When using the OTA Upgrade cluster with a non-SE profile (such as Home Automation), it is necessary to remove references to the Certicom security certificate by including the following definition:

```
#define OTA_NO_CERTIFICATE
```

20.13 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 20.13.1
2. The server and client applications must then be built - see Section 20.13.2
3. The (initial) client application must now be prepared and loaded into Flash memory of the client device - see Section 20.13.3
4. The server application must now be prepared and loaded into Flash memory of the server device - see Section 20.13.4

20.13.1 Modifying Makefiles

JN516x:

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 .ro_se_lnkKey
-j .ro_se_cert -j .ro_se_pvKey -j .ro_se_customData -j .rodata -j
.text -j .data -j .bss -j .heap -j .stack -S -O binary $< $@
```

For applications that do not use the data required for Smart Energy security (see Section 20.7.8), the following must be omitted: `.ro_se_lnkKey`, `.ro_se_cert` and `.ro_se_pvKey`.

JN5148-Z01:

In the makefiles for all applications (for server and all clients), replace the following lines:

```
$(OBJCOPY) -j .flashheader -j .oad -j .mac -j .heap_location -j
.rtc_clt  -j .rodata -j .data -j .text -j .overlay_location
$(addprefix -j .,$(OVERLAYS)) -j .bss -j .heap -j .stack -S -O
binary $< $@
```

with:

```
$(OBJCOPY) -j .bir -j .flashheader -j .heap_location -j .rtc_clt -
j .ro_ota_header -j .ro_se_lnkKey -j .ro_se_cert -j .ro_se_pvKey -
j .rodata -j .data -j .text -j .overlay_location $(addprefix -j
.,$(OVERLAYS)) -j .bss -j .heap -j .stack -S -O binary $< $@
```

For applications that do not use the data required for Smart Energy security (see Section 20.7.8), the following must be omitted: `.ro_se_lnkKey`, `.ro_se_cert` and `.ro_se_pvKey`.

Overlays (paging) must be enabled for the application by adding the following lines to the makefile:

```
ZBPRO_OVERLAYS = 1
INCFLAGS += -I$(COMPONENTS_BASE_DIR)/OVLY/Include
APPLIBS += OVLY
```

Note: Overlays must also be enabled in the RTOS configuration. In the JenOS Configuration Editor, go to the **Properties** tab, select **RTOS Configuration** and set the **Overlays** property to ‘true’.

20.13.2 Building Applications

The server and client applications must be built with the makefiles adapted for OTA upgrade (see Section 20.13.1). A build can be conducted from the command line or within the Eclipse IDE, as for any ZigBee PRO application - refer to the *SDK Installation and User Guide (JN-UG-3064)*.

The resulting binary files must then be prepared and loaded into Flash memory as described in Section 20.13.3 and Section 20.13.4.

20.13.3 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 JN51xx Flash Programmer (normally only used in a development environment) or the Atomic Programming AP-114 device.

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.

20.13.4 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 JN51xx Flash programmer (normally only used in a development environment) or the Atomic Programming AP-114 device.

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, as described in Appendix D.2.
- In a development environment, it may be loaded into Flash memory using a Flash programming tool such as the JN51xx Flash Programmer.

  However, the JN51xx 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 Jennic Encryption Tool (JET) before being loaded into the server. Use of this tool is described in the *JET User Guide (JN-UG-3081)* - the tool and its User Guide are available on request from NXP Support.

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.

21. EZ-mode Commissioning Module

This chapter describes the EZ-mode Commissioning module (EZ is pronounced ‘easy’), which can be used by an application to facilitate device commissioning.

Note: The EZ-mode Commissioning module is not strictly a part of the ZigBee Cluster Library. It is defined in the ZigBee Home Automation 1.2 profile and is currently only available for Home Automation.

21.1 Overview

The EZ-mode Commissioning module provides a means of introducing a new device into a network and pairing it for use with one or more other network nodes. This commissioning method involves user interactions, such as button-presses, on the physical devices.

To use the EZ-mode Commissioning module, you must include the file **haEzCommissioning.h** in your application and enable the module by defining the compile-time option EZ_MODE_COMMISSIONING in the **zcl_options.h** file. Other compile-time options are detailed in Section 21.8.

Note: The Identify cluster from the ZCL must also be enabled to allow a node to identify itself (e.g. by flashing a light) during commissioning. If group commissioning is required, the Groups cluster must also be enabled. The Identify cluster is described in Chapter 7 and the Groups cluster is described in Chapter 8.

21.2 Commissioning Process and Stages

The EZ-mode commissioning process consists of three basic stages/states, as follows:

1. Invocation
2. Network Steering
3. Find and Bind (or alternatively, Grouping)

These states are described in the sub-sections below. Invocation and Network Steering are also collectively referred to as ‘Set-Up’.

Note 1: Events generated during these three stages must be handled by the user-defined callback function registered on the relevant endpoint. These events are listed and described in Section 21.4.

Note 2: During EZ-mode commissioning, the current stage of commissioning of a device can be obtained using the function **eEZ_Status()**.

A set of user actions (possibly initiated by button-presses) that can be performed within the above stages have been defined by ZigBee along with recommended terminology to refer to them. These actions/terminology are:

- Join Network
- Form Network
- Allow Others To Join Network
- Restore Factory Fresh Settings
- Pair Devices
- Enable Identify Mode

The descriptions of the above actions from the Home Automation specification are provided in Appendix E.

21.2.1 Invocation

On the device to be commissioned, the application must start the commissioning process by calling the function **vEZ_SetUp()** from the main task of the application. This function call will be prompted by a user action, such as pressing a button. The function will start the ZigBee stack, if it is not already running, and then initiate the Network Steering phase (see Section 21.2.2).

> **Note:** Before **vEZ_SetUp()** is called, it is possible to change the 'Set-Up policy' (from the default one) using the function **vEZ_SetEZSetUpPolicy()**. For details, refer to the function description on page 480. The default policy is assumed here, in which a Co-ordinator will always form a new network and a Router or End Device will always search for a network to join.

The end of the 'Set-Up' phase (Invocation and Network Steering) will be indicated by the event E_EZ_SET_UP_COMPLETE. A timeout can be set for this phase through the macro EZ_SET_UP_TIME_IN_SEC (the default value is 45 seconds). If the E_EZ_SET_UP_COMPLETE event is not generated within this time after calling **vEZ_SetUp()**, the event E_EZ_SET_UP_TIMEOUT is generated instead and the commissioning is aborted.

21.2.2 Network Steering

The Network Steering stage is started by the function **vEZ_SetUp()** called during the Invocation stage (see Section 21.2.1). The objective is to join the local device to a network. Therefore, the path taken during this phase depends on whether the device is already a member of a network, as described in Section 21.2.2.1 and Section 21.2.2.2 below.

> **Note 1:** A timeout is implemented from the instant that the function **vEZ_SetUp()** is called. For more details, refer to Section 21.2.1.
>
> **Note 2:** During this phase, a default Group ID is set on any device that may later need to create a group (see Section 21.2.4). This default value is set to the 16-bit network address of the device.

21.2.2.1 Not a Network Member

If the device is not already a member of a network, the following process is followed:

1. This step depends on the ZigBee node type of the new device.

   If the device is a Co-ordinator, it will attempt to form a network. It will select an operating channel from those specified in its ZPS configuration.

   If the device is a Router or End Device:

   a) The device will perform a 'network discovery' in which it will scan the channels specified in its ZPS configuration. If configured, the 'primary' channels 11, 14, 15, 19, 20, 24 and 25 will be scanned first. If no suitable network is found in any of these channels, the device will scan any other configured channels.

   b) If only one network was found, the device will join this network.

   If multiple networks were found, the device will proceed as follows:

   - If the macro EZ_JOIN_BEST_RSSI is defined, the device will join the network with the best RSSI (Received Signal Strength Indicator) value.
   - If the above macro is undefined, the device will not join any network and the event E_EZ_MULTIPLE_OPEN_NETWORKS will be generated to inform the application.

   If no network was found, the device will continue scanning, making up to a maximum of EZ_MAX_SCAN_ATTEMPTS scan attempts (default is 3), the interval between scans being EZ_INTERSCAN_DURATION (default is 1 second). If still no network is found, the device will exit commissioning and the event E_EZ_NO_NETWORK will be generated to inform the application.

2. This step is only applicable to a Co-ordinator or Router.

   After successfully forming/joining a network, the device will enable its 'permit joining' functionality for a duration of EZ_MODE_TIME (default is 3 minutes) and will broadcast this 'permit joining' time. Thus, the device will allow other devices to join it during this time.

3. On successful completion of the Network Steering phase, the event E_EZ_SET_UP_COMPLETE is generated to inform the application on the device. If the set-up is not successful, an error will be reported using the relevant event (see Section 21.4).

Note: The parent node through which the device joins a network is not necessarily a node with which the new node will eventually be paired for operational purposes (see 'Find and Bind' in Section 21.2.3).

Signalling Progress

During the above process, it is recommended that the device signals its progress to the user by indicating when it is in the following states:

- Searching for or joining a network (in the Set-Up phase)
- Has successfully joined a network (end of the Set-Up phase)
- Must become the Co-ordinator of a new network

A range of visual or aural methods can be adopted to signal to the user, such as flashing a green light on the device.

21.2.2.2 Already a Network Member

If the device to be commissioned is already a network member when **vEZ_SetUp()** is called and is a Coordinator or Router, the device will enable its 'permit joining' functionality for a duration of EZ_MODE_TIME (default is 3 minutes) and will broadcast this 'permit joining' time. Thus, the device will allow other devices to join it during this time.

The event E_EZ_SET_UP_COMPLETE is generated on the device to inform the application of the successful completion of the Network Steering phase.

21.2.3 Find and Bind

Once a new node has been introduced into a network (as described in Section 21.2.2), the 'Find and Bind' stage allows the node to be paired with another node - for example, a new lamp may need to be paired with a controller device, to allow control of the lamp. The objective of this phase is to bind an endpoint on the new device with a compatible endpoint on an existing device in the network (depending on the supported clusters).

In the Find and Bind phase (and Grouping phase), a device can have one of two roles in EZ-mode commissioning:

- **Initiator:** This device can either request a binding with a remote endpoint or request that the remote endpoint is added to a group
- **Target:** This device receives and responds to requests from the initiator

The intended outcome is a pairing between the initiator and the target. Usually, the initiator is a controller device.

The ability of a device to perform one or both of the above commissioning roles must be configured at compile-time in the **zcl_options.h** file (see Section 21.8).

Note: During the Find and Bind phase, it is necessary to put into 'identification' mode (of the Identify cluster) all of the target devices with which the initiator will be paired. For example, if a light-switch is to control three new lamps then all three lamps must be put into identification mode (e.g. by pressing buttons).

The ‘Find and Bind’ process is as follows:

1. On the target device(s), put the devices into identification mode by calling the function **eEZ_FindAndBind()** with the option E_EZ_TARGET. This function call will be prompted by a user action, such as pressing a button. The device(s) will remain in this mode for a duration, in minutes, equal to the value of EZ_MODE_TIME.
2. On the initiator device, enter the ‘Find and Bind’ stage by calling the function **eEZ_FindAndBind()** with the option E_EZ_INITIATOR. Again, this function call will be prompted by a user action, such as pressing a button. The device will remain in this mode for a duration, in minutes, equal to the value of EZ_MODE_TIME.
3. The initiator and target devices will then exchange messages as follows:
   - **a)** The initiator will broadcast an Identify Query request and wait for Identify Query responses for a time equal to the value of EZ_RESPONSE_TIME (default is 10 seconds). If no response is received within this time, the initiator will repeatedly broadcast an Identify Query request every EZ_RESPONSE_TIME seconds until either a response is received or the EZ_MODE_TIME timeout has expired. In the latter case, the device will indicate that no device was found by generating the event E_EZ_NO_DEVICE_IN_IDENTIFY_MODE and will exit EZ-mode commissioning. The application should indicate to the user that there is no device in identification mode (e.g. by flashing an LED).
   - **b)** On receiving an Identify Query response, the initiator will check whether the IEEE address of the originating target device is already known. If this address is not known, the initiator will send an IEEE Address request to the target. On receiving the IEEE Address response, the initiator will save the address details and will send a Simple Descriptor request to the target. This must be done within the time EZ_RESPONSE_TIME from the initial Identify Query request.
   - **c)** On receiving a Simple Descriptor response, the initiator will check for client/server matches between the clusters supported by itself and the originating target device. If there is a cluster match, the initiator creates a local Binding table entry for the target/cluster. Note that a cluster can be excluded from this matching and binding process by calling the function **eEZ_ExcludeClusterFromEZBinding()** before the Find and Bind stage is started (this function can be called multiple times to exclude multiple clusters).
4. After a time EZ_MODE_TIME on each device (initiator or target), the device will exit EZ-mode commissioning and will generate the event E_EZ_FIND_AND_BIND_COMPLETE to inform the application. It is recommended that the event handler indicates the successful completion of the Find and Bind phase to the user by some visual means, such as flashing an LED.

Note 1: EZ-mode commissioning can be exited at any time using the function **vEZ_Exit()**. This function may be called as the result of a user action, such as a button-press. This is useful if all binding completes well before the EZ_MODE_TIME timeout expires.

Note 2: The EZ-mode commissioning configuration can subsequently be reset using the function **vEZ_FactoryReset()**. This will remove all Binding table entries when called on the initiator device.

21.2.4 Grouping

The ‘Grouping’ stage is an alternative to the ‘Find and Bind’ stage, and also employs an initiator device and target devices (as described in Section 21.2.3). Grouping is recommended instead of Find and Bind when the initiator device needs to be bound to more than five target devices. In this case, the targets are assigned a group address which, during normal operation, will be used to broadcast to all the targets (rather than unicast to the individual targets).

Note 1: The Grouping feature requires the Groups cluster to be enabled on the participating devices. The Groups cluster is described in Chapter 8.

Note 2: During the Grouping phase, it is necessary to put into ‘identification’ mode (of the Identify cluster) all of the nodes with which the initiator will be paired. For example, if a new light-switch is to control six lamps then all six lamps must be put into identification mode (e.g. by pressing buttons).

Note 3: During the Network Steering phase, a default Group ID is set on any device which can become an initiator and may need to create a group. This default value is set to the 16-bit network address of the device.

To use the Grouping feature, the macro EZ_ENABLE_GROUP must be defined in the **zcl_options.h** file on the initiator and target devices.

The ‘Grouping’ process is as follows:

1. On the target device(s), put the devices into identification mode by calling the function **eEZ_Group()** with the option E_EZ_TARGET. This function call will be prompted by a user action, such as pressing a button. The device(s) will remain in this mode for a duration, in minutes, equal to the value of EZ_MODE_TIME.

2. On the initiator device, enter the ‘Grouping’ stage by calling the function **eEZ_Group()** with the option E_EZ_INITIATOR. Again, this function call will be prompted by a user action, such as pressing a button. The device will remain in this mode for a duration, in minutes, equal to the value of EZ_MODE_TIME.

> **Note:** If a custom Group ID is to used (instead of the default Group ID set during the Network Steering phase) then this should be set by calling the function **vEZ_SetGroupId()** on the initiator before **eEZ_Group()**.

3. The initiator and target devices will then exchange messages as follows:

   a) The initiator will broadcast an Identify Query request and wait for Identify Query responses for a time equal to the value of EZ_RESPONSE_TIME (default is 10 seconds). If no response is received within this time, the initiator will repeatedly broadcast an Identify Query request every EZ_RESPONSE_TIME seconds until either a response is received or the EZ_MODE_TIME timeout has expired. In the latter case, the device will indicate that no device was found by generating the event E_EZ_NO_DEVICE_IN_IDENTIFY_MODE and will exit EZ-mode commissioning. The application should indicate to the user that there is no device in identification mode (e.g. by flashing an LED).

   b) On receiving an Identify Query response, the initiator will check whether the IEEE address of the originating target device is already known. If this address is not known, the initiator will send an IEEE Address request to the target. On receiving the IEEE Address response, the initiator will save the address details and will send a Simple Descriptor request to the target. This must be done within the time EZ_RESPONSE_TIME from the initial Identify Query request.

   c) On receiving a Simple Descriptor response, the initiator will check for client/server matches between the clusters supported by itself and the originating target device. If there is a cluster match, the initiator sends an 'Add Group If Identifying' command to the target device. The event E_EZ_GROUP_CREATED_FOR_TARGET is also generated. The initiator identifies the group using either its default Group ID or, if specified through a call to **vEZ_SetGroupId()**, a custom Group ID.

   > **Note:** On generation of the event E_EZ_GROUP_CREATED_FOR_TARGET, the application on the initiator can optionally call the function **eCLD_IdentifyCommandIdentifyRequestSend()** of the Identify cluster in order to request the grouped target device to exit identification mode.

   d) On receiving an 'Add Group If Identifying' command, a target device will add the group into its Group table.

e) The initiator will remain in this mode for EZ_MODE_TIME and repeatedly broadcast an Identify Query request every EZ_RESPONSE_TIME seconds until the EZ_MODE_TIME timeout has expired.

4. After a time EZ_MODE_TIME on each device (initiator or target), the device will exit EZ-mode commissioning and will generate the event E_EZ_GROUPING_COMPLETE to inform the application. It is recommended that the event handler indicates the successful completion of the Grouping phase to the user by some visual means, such as flashing an LED.

Note 1: EZ-mode commissioning can be exited at any time using the function **vEZ_Exit()**. This function may be called as the result of a user action, such as a button-press. This is useful if all grouping completes well before the EZ_MODE_TIME timeout expires.

Note 2: The EZ-mode commissioning configuration can subsequently be reset using the function **vEZ_FactoryReset()**. This will remove all Group table entries when called on a target device and will clear the group address when called on the initiator device.

21.3 Persisting Commissioning Data

It is important to persist commissioning data by saving it in non-volatile memory on the local device, so that commissioned bindings and/or groupings are not lost during a power outage or sleep without RAM held. This data preservation is normally handled by the JenOS Persistent Data Manager (PDM). Binding tables and Group tables will be saved and recovered by PDM, but two EZ-mode commissioning functions are provided to facilitate the preservation of other commissioning data:

- **vEZ_StoreEZState()** can be used to pass commissioning data to PDM for storage.
- **vEZ_UpdateEZState()** can then be used to restore the preserved commissioning data following a power outage or sleep without RAM held. This function should be called after PDM has recovered the data from non-volatile memory.

This commissioning data includes a status value and, if applicable, the group address associated with the initiator. This data is held in a structure which is filled in automatically and is not the concern of the application.

21.4 EZ-mode Commissioning Events

The events that can be generated during EZ-mode commissioning are defined in the structure `teZCL_CallBackEventType` (see Section). They are pre-fixed with E_EZ and are listed in the table below, which also indicates the device types (Co-ordinator or Router and End Device or all device types) on which the events can occur.

Device Types	Event
Co-ordinator	E_EZ_FAILED_TO_START
Router and End Device	E_EZ_NO_NETWORK *
	E_EZ_MULTIPLE_OPEN_NETWORKS *
	E_EZ_FAILED_TO_JOIN *
	E_EZ_ZDO_JOIN_API_FAILED *
All device types	E_EZ_ZDO_START_API_FAILED
	E_EZ_NO_DEVICE_IN_IDENTIFY_MODE
	E_EZ_SET_UP_COMPLETE
	E_EZ_SET_UP_TIMEOUT
	E_EZ_BIND_CREATED_FOR_TARGET
	E_EZ_GROUP_CREATED_FOR_TARGET
	E_EZ_BIND_FAILED
	E_EZ_FIND_AND_BIND_COMPLETE
	E_EZ_GROUPING_COMPLETE

Table 20: EZ-mode Commissioning Events

* When the alternative ‘Set-Up policy’ E_EZ_JOIN_ELSE_FORM_IF_NO_NETWORK is used, these events can also be generated on a Co-ordinator.

The EZ-mode commissioning events are ZCL events. Therefore, 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 below.

E_EZ_FAILED_TO_START

This event is generated when **vEZ_SetUp()** has been called on a Co-ordinator but the device fails to start a new network.

E_EZ_NO_NETWORK

This event is generated when **vEZ_SetUp()** has been called on a Router or End Device but no network is available to join. If the first scan detects no available network, the scan is repeated up to EZ_MAX_SCAN_ATTEMPTS times. If still no available network is detected, this event is generated.

E_EZ_MULTIPLE_OPEN_NETWORKS

This event is generated when **vEZ_SetUp()** has been called on a Router or End Device and more than one network is available to join. The event is generated to notify the application in order to prevent the device from joining the wrong network.

E_EZ_FAILED_TO_JOIN

This event is generated when **vEZ_SetUp()** has been called on a Router or End Device but the device fails to join a network.

E_EZ_ZDO_JOIN_API_FAILED

This event is generated on a Router or End Device when a call to the stack function **ZPS_eAplZdoJoinNetwork()** fails.

E_EZ_ZDO_START_API_FAILED

This event is generated on any device type when a call to the stack function **ZPS_eAplZdoStartStack()** fails.

E_EZ_NO_DEVICE_IN_IDENTIFY_MODE

This event is generated when **eEZ_FindAndBind()** has been called on a device but no Identify Query responses were received (therefore there were no target devices in identification mode).

E_EZ_SET_UP_COMPLETE

This event is generated on the successful completion of the Set-Up phase - that is, the device has successfully joined or formed a network.

E_EZ_SET_UP_TIMEOUT

This event is generated when the event E_EZ_SET_UP_COMPLETE is not generated within the timeout period EZ_SET_UP_TIME_IN_SEC.

E_EZ_BIND_CREATED_FOR_TARGET

This event is generated during the Find and Bind phase when the device has successfully completed a binding to a target node. The application can access the details of the bound device through the structure `tsZCL_EZModeBindDetails` (see Section 21.7.1), which is a field of the event structure `tsZCL_CallBackEvent`.

E_EZ_GROUP_CREATED_FOR_TARGET

This event is generated during the Grouping phase on an initiator device when it adds a target node to a group. The application can access the details of the grouped device through the structure `tsZCL_EZModeGroupDetails` (see Section 21.7.2), which is a field of the event structure `tsZCL_CallBackEvent`.

E_EZ_BIND_FAILED

This event is generated during the Find and Bind phase when an attempted binding to a target device has failed.

E_EZ_FIND_AND_BIND_COMPLETE

This event is generated on the successful completion of the Find and Bind phase.

E_EZ_GROUPING_COMPLETE

This event is generated on the successful completion of the Grouping phase.

21.5 Functions

The following EZ-mode commissioning functions are provided in the HA API:

Function	**Page**
vEZ_SetUp	471
eEZ_FindAndBind	472
eEZ_ExcludeClusterFromEZBinding	473
eEZ_Group	474
vEZ_SetGroupId	475
vEZ_Exit	476
eEZ_Status	477
vEZ_StoreEZState	478
vEZ_UpdateEZState	479
vEZ_SetEZSetUpPolicy	480
vEZ_FactoryReset	481

vEZ_SetUp

```
void vEZ_SetUp(void);
```

Description

This function is used to start the 'Set-Up' phase (Invocation and Network Steering) of EZ-mode commissioning on the device to be commissioned. It must be called from the main task of the application on the device.

If the device is not already a member of a network, the function will start the ZigBee stack (if necessary) and initiate a 'network discovery', after which the device will join a network (if a Router or End Device) or form a network (if a Co-ordinator).

If the device is a Co-ordinator or Router, once the device is a member of a network, the function will allow other nodes to join the local node during a time interval equal to the value of EZ_MODE_TIME (default is 3 minutes).

The function is non-blocking and returns immediately. The successful completion of the Set-Up phase is indicated by the event E_EZ_SET_UP_COMPLETE.

A timeout is applied to this phase and its value, in seconds, can be set through the macro EZ_SET_UP_TIME_IN_SEC (the default value is 45 seconds). If the E_EZ_SET_UP_COMPLETE event is not generated within this time after calling **vEZ_SetUp()**, the event E_EZ_SET_UP_TIMEOUT is generated instead and the commissioning is aborted.

For more details of the use of this function in the Invocation and Network Steering phases of EZ-mode commissioning, refer to Section 21.2.1 and Section 21.2.2.

Parameters

None

Returns

None

eEZ_FindAndBind

```
teZCL_Status eEZ_FindAndBind(uint8 u8SourceEndpoint,
                             teEZ_Mode eEZMode);
```

Description

This function is used to start the 'Find and Bind' phase of EZ-mode commissioning on the initiator device or a target device:

- On the initiator device, the function must be called with the option E_EZ_INITIATOR. The function enables the initiator to send requests in order to find suitable endpoints with which to pair and to perform this pairing.
- On a target device, the function must be called with the option E_EZ_TARGET. The function puts the device into 'identification' mode (of the Identify cluster) and enables the device to respond to requests from an initiator device.

In both cases, the function call will be prompted by a user action, such as pressing a button. The device will remain in this mode for a duration, in seconds, equal to the value of EZ_MODE_TIME (default is 3 minutes).

The function is non-blocking and returns immediately. After a time EZ_MODE_TIME (on each device) following this function call, the device will exit EZ-mode commissioning and will generate the event E_EZ_FIND_AND_BIND_COMPLETE to inform the application.

For more details of the use of this function in the Find and Bind phase of EZ-mode commissioning, refer to Section 21.2.3.

Parameters

u8SourceEndpoint	Number of endpoint on which this function is called
eEZMode	Type of commissioning node (initiator or target) on which this function is called, one of: E_EZ_INITIATOR E_EZ_TARGET

Returns

E_ZCL_FAIL

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

eEZ_ExcludeClusterFromEZBinding

```
teEZ_ClusterExcludeStatus eEZ_ExcludeClusterFromEZBinding(
                                          uint16 u16ClusterID);
```

Description

This function can be called on the initiator to exclude the specified cluster from the binding process during the Find and Bind phase. During this phase, the initiator will bind with any endpoint (on a target device) with a suitable client/server cluster match. If it is not appropriate to include a particular cluster (even if a match exists), the cluster can be excluded from the process using this function. This allows the use of the local Binding table to be optimised.

If this function is required, it must be called before the Find and Bind phase is started using **eEZ_FindAndBind()**.

If more than one cluster needs to be excluded, the function can be called multiple times. The function internally stores an array of clusters that are excluded from binding. The array size is configurable using the macro EZ_MAX_CLUSTER_EXCLUSION_SIZE (the default is 5). If an attempt is made to exceed this limit, the function will return E_EZ_EXCLUSION_TABLE_FULL.

Parameters

u16ClusterID	Cluster ID of cluster to be excluded

Returns

E_EZ_CLUSTER_EXCLUSION_SUCCESS

E_EZ_EXCLUSION_TABLE_FULL

eEZ_Group

```
teZCL_Status eEZ_Group(uint8 u8SourceEndpoint,
                       eEZ_Mode eEZMode);
```

Description

This function is used to start the ‘Grouping’ phase of EZ-mode commissioning on the initiator device or a target device:

- On the initiator device, the function must be called with the option E_EZ_INITIATOR. The function enables the initiator to send requests in order to find target endpoints with which to pair and collect into a group.
- On a target device, the function must be called with the option E_EZ_TARGET. The function puts the device into ‘identification’ mode (of the Identify cluster) and enables the device to respond to requests/commands from an initiator device.

In both cases, the function call will be prompted by a user action, such as pressing a button. The device will remain in this mode for a duration, in seconds, equal to the value of EZ_MODE_TIME (default is 3 minutes).

The function is non-blocking and returns immediately. After a time EZ_MODE_TIME (on each device) following this function call, the device will exit EZ-mode commissioning and will generate the event E_EZ_GROUPING_COMPLETE to inform the application.

For more details of the use of this function in the Grouping phase of EZ-mode commissioning, refer to Section 21.2.4.

Parameters

u8SourceEndpoint	Number of endpoint on which this function is called
eEZMode	Type of commissioning node (initiator or target) on which this function is called, one of: E_EZ_INITIATOR E_EZ_TARGET

Returns

E_ZCL_FAIL
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

vEZ_SetGroupId

```
void vEZ_SetGroupId(uint16 u16GroupID);
```

Description

This function can be used on the initiator to specify a Group ID which will be used in the 'Grouping' phase of EZ-mode commissioning. The specified16-bit identifier will be allocated to the group that is created when **eEZ_Group()** is called.

If required, the **vEZ_SetGroupId()** function must be called before **eEZ_Group()** at the start of the Grouping phase. It may be required in either of the following circumstances:

- A custom Group ID is to used instead of the default Group ID which was set during the Network Steering phase of EZ-mode commissioning (this default Group ID was set to the 16-bit network address of the device when it joined or formed the network)
- A custom Group ID is required because the device did not join or form the network via EZ-mode commissioning and therefore has no default Group ID

Parameters

u16GroupID	16-bit Group ID to be assigned to group

Returns

None

vEZ_Exit

```
void vEZ_Exit(uint8 u8SourceEndpoint);
```

Description

This function can be used to exit EZ-mode commissioning. This is likely to be as the result of a user action such as a button-press. The function is useful during the ‘Find and Bind’ or ‘Grouping’ stage to avoid waiting for the EZ_MODE_TIME timeout to expire - for example, if there are few nodes to bind or group and the binding/grouping operation is completed well before the timeout.

Parameters

u8SourceEndpoint	Number of endpoint on which this function is called

Returns

None

eEZ_Status

```
teEZ_Status eEZ_Status(uint8 u8SourceEndpoint);
```

Description

This function can be used during EZ-mode commissioning to request the status of the commissioning process on the local device. It returns the EZ-mode commissioning stage that the device is currently in - one of:

- Set-Up (Invocation and Network Steering)
- Find and Bind
- Grouping
- Idle

Parameters

u8SourceEndpoint	Number of endpoint on which this function is called

Returns

E_EZ_IDLE
E_EZ_SETUP_IN_PROGRESS
E_EZ_FIND_AND_BIND_IN_PROGRESS
E_EZ_GROUPING_IN_PROGRESS

vEZ_StoreEZState

```
void vEZ_StoreEZState(
                        tsEZ_PersistentData *psEZ_PDMData);
```

Description

This function can be used to request the specified EZ-mode commissioning data to be preserved in non-volatile memory on the local device so that the data can be recovered following a power outage or sleep without RAM held. Persistent data of this kind is managed by the JenOS PDM module. This function passes the data to PDM which will then save it in non-volatile memory.

The commissioning data to be persisted is provided in a structure in which the data is automatically inserted (populating this structure is not the responsibility of the application).

Persisting commissioning data is described in Section 21.3.

Parameters

psEZ_PDMData	Pointer to persistent data structure (see Section 21.7.3) containing the data to be saved

Returns

None

vEZ_UpdateEZState

```
void vEZ_UpdateEZState(
                        tsEZ_PersistentData *psEZ_PDMData);
```

Description

This function can be used to restore EZ-mode commissioning data which has been retrieved from non-volatile memory on the local device. The JenOS PDM module will recover the data from non-volatile memory following a power outage or sleep without RAM held. This function should be called only after PDM has read the data from non-volatile memory.

Persisting commissioning data is described in Section 21.3.

Parameters

psEZ_PDMData	Pointer to persistent data structure (see Section 21.7.3) to receive the retrieved data

Returns

None

vEZ_SetEZSetUpPolicy

```
void vEZ_SetEZSetUpPolicy(eEZ_SetUpPolicy ePolicy);
```

Description

This function can be used to set the commissioning policy on a device before **vEZ_SetUp()** is called. The possible policies are as follows:

- E_EZ_JOIN_OR_FORM_BASED_ON_DEVICE_TYPE (default): A Co-ordinator device will always form a network. A Router or End Device will always search for a suitable network to join - if no network is available, the device will exit Set-Up and inform the application via the event E_EZ_NO_NETWORK.
- E_EZ_JOIN_ELSE_FORM_IF_NO_NETWORK: A Co-ordinator device will first search for a suitable network to join. If no network is available, the device will inform the application via the event E_EZ_NO_NETWORK. On the next call to **vEZ_SetUp()**, the device will form a network.

Since the first policy above is used by default, a call to this function is only required if the second policy is to be adopted.

Parameters

ePolicy	Set-Up policy to use (see above), one of: E_EZ_JOIN_OR_FORM_BASED_ON_DEVICE_TYPE E_EZ_JOIN_ELSE_FORM_IF_NO_NETWORK

Returns

None

vEZ_FactoryReset

```
void vEZ_FactoryReset(uint8 u8SourceEndpoint);
```

Description

This function is used to reset the EZ-mode commissioning configuration on the local node.

- It will remove all Binding table entries when called on the initiator device
- If the ‘Grouping’ feature is enabled, it will remove all Group table entries when called on the target devices and will clear the group address when called on the initiator device

Parameters

u8SourceEndpoint	Number of endpoint on which this function is called

Returns

None

21.6 Enumerations

21.6.1 'Set-Up Policy' Enumerations

The following enumerations are used to specify the 'Set-Up policy' to use.

```
typedef enum
{
   E_EZ_JOIN_OR_FORM_BASED_ON_DEVICE_TYPE,
   E_EZ_JOIN_ELSE_FORM_IF_NO_NETWORK
}eEZ_SetUpPolicy;
```

The enumerations are described in the table below.

Enumeration	Description
E_EZ_JOIN_OR_FORM_BASED_ON_DEVICE_TYPE	A Co-ordinator device will always form a network. A Router or End Device will always search for a suitable network to join - if no network is available, the device will exit Set-Up and inform the application via the event E_EZ_NO_NETWORK. This is the default policy.
E_EZ_JOIN_ELSE_FORM_IF_NO_NETWORK	A Co-ordinator device will first search for a suitable network to join. If no network is available, the device will inform the application via the event E_EZ_NO_NETWORK. On the next call to **vEZ_SetUp()**, the device will form a network.

Table 21: 'Set-Up Policy' Enumerations

21.6.2 Status Enumerations (Return Codes)

The following enumerations are used to indicate the current stage of EZ-mode commissioning.

```
Typedef enum
{
   E_EZ_IDLE,
   E_EZ_SETUP_IN_PROGRESS,
   E_EZ_FIND_AND_BIND_IN_PROGRESS,
   E_EZ_GROUPING_IN_PROGRESS
} eEZ_Status;
```

The enumerations are described in the table below.

Enumeration	Description
E_EZ_IDLE	No commissioning in progress
E_EZ_SETUP_IN_PROGRESS	Set-Up stage (Invocation and Network Steering)
E_EZ_FIND_AND_BIND_IN_PROGRESS	Find and Bind stage
E_EZ_GROUPING_IN_PROGRESS	Grouping stage

Table 22: Status Enumerations

21.6.3 ‘Cluster Exclude’ Enumerations

The following enumerations are used to indicate the outcome of an attempt to exclude a cluster from the binding process.

```
typedef enum
{
   E_EZ_CLUSTER_EXCLUSION_SUCCESS,
   E_EZ_EXCLUSION_TABLE_FULL
}teEZ_ClusterExcludeStatus;
```

The enumerations are described in the table below.

Enumeration	Description
E_EZ_CLUSTER_EXCLUSION_SUCCESS	Cluster was successfully excluded
E_EZ_EXCLUSION_TABLE_FULL	Cluster was not excluded because the ‘exclusion table’ is full - the number of entries has reached the limit set by the macro EZ_MAX_CLUSTER_EXCLUSION_SIZE

Table 23: ‘Cluster Exclude’ Enumerations

21.6.4 Event Enumerations

Thee following enumerations represent the EZ-mode commissioning events.

```
typedef enum
{
   E_EZ_NO_NETWORK = E_ZCL_CBET_ENUM_END + 1,
   E_EZ_ZDO_START_API_FAILED,
   E_EZ_ZDO_JOIN_API_FAILED,
   E_EZ_MULTIPLE_OPEN_NETWORKS,
   E_EZ_FAILED_TO_START,
   E_EZ_FAILED_TO_JOIN,
   E_EZ_SET_UP_TIMEOUT,
   E_EZ_SET_UP_COMPLETE,
   E_EZ_NO_DEVICE_IN_IDENTIFY_MODE,
   E_EZ_BIND_CREATED_FOR_TARGET,
   E_EZ_GROUP_CREATED_FOR_TARGET,
   E_EZ_BIND_FAILED,
   E_EZ_FIND_AND_BIND_COMPLETE,
   E_EZ_GROUPING_COMPLETE,
   E_EZ_NONE
}teEZ_Events;
```

The EZ-mode commissioning events are described in Section 21.4.

21.7 Structures

21.7.1 tsZCL_EZModeBindDetails

This structure contains the details of a binding made with a cluster on an endpoint of a target device.

```
typedef struct
{
   uint8      u8DstEndpoint;
   uint16     u16ClusterID;
   uint16     u16DstShortAddress;
}tsZCL_EZModeBindDetails;
```

where:

- `u8DstEndpoint` is the number of the endpoint (on the target device) with which the binding has been made
- `u16ClusterID` is the Cluster ID of the cluster for which a client/server match has been found
- `u16DstShortAddress` is the 16-bit network address of the target device

21.7.2 tsZCL_EZModeGroupDetails

This structure contains the details of the addition of a remote endpoint to a group.

```
typedef struct
{
    uint8     u8DstEndpoint;
    uint16    u16GroupID;
    uint16    u16DstShortAddress;
}tsZCL_EZModeGroupDetails;
```

where:

- `u8DstEndpoint` is the number of the endpoint (on the target device) which has been added to a group
- `u16GroupID` is the Group ID of the group to which the endpoint (on the target device) has been added
- `u16DstShortAddress` is the 16-bit network address of the target device which contains the grouped endpoint

21.7.3 tsEZ_PersistentData

This structure contains commissioning data that is to be persisted in non-volatile memory.

```
typedef struct
{
   uint8      u8EZState;
   uint16     u16GroupID;
}tsEZ_PersistentData;
```

This structure is automatically filled in and no knowledge of its contents is required.

21.8 Compile-Time Options

This section describes the compile-time options that may be selected in the **zcl_options.h** file of an application that uses the EZ-mode Commissioning module.

To enable the EZ-mode Commissioning module in the code to be built, it is necessary to add the following to the above header file:

```
#define EZ_MODE_COMMISSIONING
```

It is also necessary to add one or both of the following lines to the file, depending on whether the device can be an initiator or a target during the ‘Find and Bind’ and ‘Grouping’ phases of commissioning:

```
#define EZ_MODE_INITIATOR
#define EZ_MODE_TARGET
```

The EZ-mode Commissioning module contains macros that may be optionally specified at compile-time by adding some or all the following lines to the **zcl_options.h** file.

Maximum number of scan attempts

The maximum number of scan attempts performed by the initiator during the Network Steering phase can be set (to n) by including the following line:

```
#define EZ_MAX_SCAN_ATTEMPTS n
```

The default value is 3.

Inter-scan duration

The time-interval, in seconds, between consecutive scan attempts during the Network Steering phase can be set (to `t`) by including the following line:

```
#define EZ_INTERSCAN_DURATION t
```

The default value is 1 second.

Timeout for ‘Identify Query’ response

The maximum time, in seconds, for which the initiator will wait for an Identify Query response (after broadcasting an Identify Query request) can be set (to `t`) by including the following line:

```
#define EZ_RESPONSE_TIME t
```

The default value is 10 seconds.

Enable RSSI for joining

Use of the RSSI value for deciding which network to join (the device will join the network with the best RSSI) can be enabled by including the following line:

```
#define EZ_JOIN_BEST_RSSI
```

Maximum number of networks to consider

The maximum number of network descriptors that will be considered when choosing a network to join can be set (to n) by including the following line:

```
#define EZ_MAX_NETWORK_DESCRIPTOR n
```

The default value is 8.

This feature is only valid when EZ_JOIN_BEST_RSSI is defined.

Timeout for ‘Permit Joining’ and ‘Find and Bind’

The maximum time, in minutes, for which a device can enable ‘Permit Joining’ or spend in the ‘Find and Bind’ state can be set (to `t`) by including the following line:

```
#define EZ_MODE_TIME t
```

The default value is 3 minutes.

Maximum number of target devices for binding

The maximum number of target devices to which the initiator can be bound can be set (to n) by including the following line:

```
#define EZ_MAX_TARGET_DEVICE n
```

The default value is 10.

Maximum number of clusters excluded from binding

The maximum number of clusters that can be excluded from cluster client/server matching in the binding process can be set (to n) by including the following line:

```
#define EZ_MAX_CLUSTER_EXCLUSION_SIZE n
```

The default value is 5.

Enable Grouping

The Grouping stage can be enabled (to replace the Find and Bind stage) by including the following line:

```
#define EZ_ENABLE_GROUP
```

Timeout for stack event after a function call

The maximum time, in seconds, for which a device will wait for a stack event following a function call can be set (to `t`) by including the following line:

```
#define EZ_SET_UP_TIME_IN_SEC t
```

The default value is 45 seconds.

Maximum number of endpoints

The maximum number of endpoints supported on the local device can be set (to n) by including the following line:

```
#define EZ_NUMBER_OF_ENDPOINTS n
```

The default value is 1.

Part III: General Reference Information

22. 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 22.1
- Attribute access functions - see Section 22.2

22.1 General Functions

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

Function	Page
eZCL_Register	492
vZCL_EventHandler	493
eZCL_GetLastZpsError	494

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 (e.g. IPD of the SE profile) on an endpoint - in this case, the appropriate device registration function should be used.

The use of custom endpoints with the Smart Energy profile is described in the *Smart Energy API User Guide (JN-UG-3059)*.

Parameters

psEndPointDefinition	Pointer to `tsZCL_EndPointDefinition` structure for the endpoint to be registered (see Section 23.1.1)

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

vZCL_EventHandler

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

Description

This function should be called when an event (ZigBee stack, peripheral or cluster event) occurs. The function is used to pass the event to the ZCL. The ZCL will then process 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 23.2 for details of this structure.

An example of using the **vZCL_EventHandler()** function is provided in the Application Note *Smart Energy HAN Solutions (JN-AN-1135)*.

Parameters

psZCLCallBackEvent	Pointer to a `tsZCL_CallBackEvent` event structure (see Section 23.2) containing the event to process

Returns

None

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 Smart Energy function **eSE_ReadMeterAttributes()** returns E_ZCL_ERR_ZTRANSMIT_FAIL (because the ZigBee PRO API function that was used to transmit the request 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 will 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 PRO Stack User Guide (JN-UG-3048)*

22.2 Attribute Access Functions

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

Function	Page
eZCL_SendReadAttributesRequest	496
eZCL_SendWriteAttributesRequest	498
eZCL_SendWriteAttributesNoResponseRequest	500
eZCL_SendWriteAttributesUndividedRequest	502
eZCL_SendDiscoverAttributesRequest	504
eZCL_SendConfigureReportingCommand	506
eZCL_SendReadReportingConfigurationCommand	508
eZCL_ReportAllAttributes	510
eZCL_CreateLocalReport	511
eZCL_SetReportableFlag	512
eZCL_ReadAllAttributes	513
eZCL_HandleReadAttributesResponse	515
eZCL_ReadLocalAttributeValue	516
eZCL_WriteLocalAttributeValue	518
eZCL_OverrideClusterControlFlags	520
eZCL_SetSupportedSecurity	521

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

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. Note that read access to cluster attributes on the remote node must be enabled at compile-time as described in Section 1.2.

You must specify the endpoint on the local node from which the request is to be sent. This is also used to identify the instance of the local shared device structure which holds the relevant attributes. The obtained attribute values will be written to this shared structure by the function.

You must also specify the address of the destination node, 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 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 read selected attributes from the remote cluster. You are required to 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 manufacturer-specific or as defined in the relevant ZigBee-defined application profile.

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

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. Note that 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 will be sent
u8DestinationEndPointId	Number of the remote endpoint to which the request will be sent. Note that this parameter is ignored when sending to address types 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 23.1.4) containing the address of the remote node to which the request will be sent
pu8TransactionSequenceNumber	Pointer to a location to store the Transaction Sequence Number (TSN) of the request
u8NumberOfAttributesInRequest	Number of attributes to be read
bIsManufacturerSpecific	Indicates whether attributes are manufacturer-specific or as defined in relevant ZigBee profile: TRUE: Attributes are manufacturer-specific FALSE: Attributes are from ZigBee profile
u16ManufacturerCode	ZigBee Alliance code for the manufacturer that defined proprietary attributes (set to zero if attributes are from the ZigBee-defined profile - 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 means of enumerations (listed in the 'Enumerations' section of each cluster-specific chapter)

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

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,
                        uint16 *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 1.2.

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 application must write the new attribute values to this shared structure before calling this function - the function will then pick up these values from the shared structure before sending them to the remote endpoint.

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 manufacturer-specific or as defined in the relevant ZigBee-defined application profile.

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

Following 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 appropriate endpoint registration function (e.g. from the Smart Energy, Home Automation or ZigBee Light Link library).

Parameters

u8SourceEndPointId	Number of the local endpoint through which the request will be sent
u8DestinationEndPointId	Number of the remote endpoint to which the request will be sent. Note that this parameter is ignored when sending to address types 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 23.1.4) containing the address of the remote node to which the request will be sent
pu8TransactionSequenceNumber	Pointer to a location to store the Transaction Sequence Number (TSN) of the request
u8NumberOfAttributesInRequest	Number of attributes to be written
bIsManufacturerSpecific	Indicates whether attributes are manufacturer-specific or as defined in relevant ZigBee profile: TRUE: Attributes are manufacturer-specific FALSE: Attributes are from application profile
u16ManufacturerCode	ZigBee Alliance code for the manufacturer that defined proprietary attributes (set to zero if attributes are from the ZigBee-defined profile - that is, if *bIsManufacturerSpecific* is set to FALSE)
pu16AttributeRequestList	Pointer to an array which lists the attributes to be written. The attributes are identified by means of enumerations (listed in the 'Enumerations' section of each cluster-specific chapter)

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_RO
E_ZCL_ERR_ATTRIBUTES_ACCESS
E_ZCL_ERR_ATTRIBUTE_NOT_FOUND
E_ZCL_ERR_PARAMETER_NULL
E_ZCL_ERR_PARAMETER_RANGE

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,
                    uint16 *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 1.2.

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 application must write the new attribute values to this shared structure before calling this function - the function will then pick up these values from the shared structure before sending them to the remote endpoint.

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.

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 manufacturer-specific or as defined in the relevant ZigBee-defined application profile.

You are also required to 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 will be sent
u8DestinationEndPointId	Number of the remote endpoint to which the request will be sent. Note that this parameter is ignored when sending to address types 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 23.1.4) containing the address of the remote node to which the request will be sent
pu8TransactionSequenceNumber	Pointer to a location to store the Transaction Sequence Number (TSN) of the request
u8NumberOfAttributesInRequest	Number of attributes to be written
bIsManufacturerSpecific	Indicates whether attributes are manufacturer-specific or as defined in relevant ZigBee profile: TRUE: Attributes are manufacturer-specific FALSE: Attributes are from ZigBee profile
u16ManufacturerCode	ZigBee Alliance code for the manufacturer that defined proprietary attributes (set to zero if attributes are from the ZigBee-defined profile - that is, if *bIsManufacturerSpecific* is set to FALSE)
pu16AttributeRequestList	Pointer to an array which lists the attributes to be written. The attributes are identified by means of enumerations (listed in the 'Enumerations' section of each cluster-specific chapter)

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_RO

E_ZCL_ERR_ATTRIBUTES_ACCESS

E_ZCL_ERR_ATTRIBUTE_NOT_FOUND

E_ZCL_ERR_PARAMETER_NULL

E_ZCL_ERR_PARAMETER_RANGE

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,
                    uint16 *pu16AttributeRequestList);
```

Description

This function can be used to send an ‘undivided write attributes’ request to a cluster on a remote endpoint. This requests that all the specified attributes are updated on the remote endpoint or none at all - that is, if one of the specified attributes cannot be written then none of the attributes are updated. The function also demands a ‘write attributes’ response from the remote endpoint, indicating success or failure. Note that write access to cluster attributes on the remote node must be enabled at compile-time as described in Section 1.2.

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 application must write the new attribute values to this shared structure before calling this function - the function will then pick up these values from the shared structure before sending them to the remote endpoint.

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 manufacturer-specific or as defined in the relevant ZigBee-defined application profile.

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

Following 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 (e.g. from the Smart Energy, Home Automation or ZigBee Light Link library).

Parameters

u8SourceEndPointId	Number of the local endpoint through which the request will be sent
u8DestinationEndPointId	Number of the remote endpoint to which the request will be sent. Note that this parameter is ignored when sending to address types 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 23.1.4) containing the address of the remote node to which the request will be sent
pu8TransactionSequenceNumber	Pointer to a location to store the Transaction Sequence Number (TSN) of the request
u8NumberOfAttributesInRequest	Number of attributes to be written
bIsManufacturerSpecific	Indicates whether attributes are manufacturer-specific or as defined in relevant ZigBee profile: TRUE: Attributes are manufacturer-specific FALSE: Attributes are from ZigBee profile
u16ManufacturerCode	ZigBee Alliance code for the manufacturer that defined proprietary attributes (set to zero if attributes are from the ZigBee-defined profile - that is, if *bIsManufacturerSpecific* is set to FALSE)
pu16AttributeRequestList	Pointer to an array which lists the attributes to be written. The attributes are identified by means of enumerations (listed in the 'Enumerations' section of each cluster-specific chapter)

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_RO
E_ZCL_ERR_ATTRIBUTES_ACCESS
E_ZCL_ERR_ATTRIBUTE_NOT_FOUND
E_ZCL_ERR_PARAMETER_NULL
E_ZCL_ERR_PARAMETER_RANGE

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 will return immediately and the results of the request will later be received in a 'discover attributes' 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 will be set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

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 will 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 2.2.3.

Parameters

u8SourceEndPointId	Number of the local endpoint through which the request will be sent
u8DestinationEndPointId	Number of the remote endpoint to which the request will be sent
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 23.1.4) containing the address of the remote node to which the request will be sent
pu8TransactionSequenceNumber	Pointer to a location to store the Transaction Sequence Number (TSN) of the request
u16AttributeId	Identifier of 'start' attribute of interest
bIsManufacturerSpecific	Indicates whether attributes are manufacturer-specific or as defined in relevant ZigBee profile: TRUE: Attributes are manufacturer-specific FALSE: Attributes are from ZigBee profile
u16ManufacturerCode	ZigBee Alliance code for the manufacturer that defined proprietary attributes (set to zero if attributes are from the ZigBee-defined profile - 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

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 will be set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

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 will be sent
u8DestinationEndPointId	Number of the remote endpoint to which the request will be sent
u16ClusterId	Identifier of the cluster to be configured (see the macros section in the cluster header file)
psDestinationAddress	Pointer to a structure (see Section 23.1.4) containing the address of the remote node to which the request will be sent
pu8TransactionSequenceNumber	Pointer to a location to store the Transaction Sequence Number (TSN) of the request
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 profile: TRUE: Attributes are manufacturer-specific FALSE: Attributes are from ZigBee profile
u16ManufacturerCode	ZigBee Alliance code for the manufacturer that defined proprietary attributes (set to zero if attributes are from the ZigBee-defined profile - 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 23.1.5)

Returns

E_ZCL_SUCCESS

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 will be set to match the TSN in the request, allowing an incoming response to be paired with a request. This is useful when sending more than one request to the same destination endpoint.

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 will be sent
u8DestinationEndPointId	Number of the remote endpoint to which the request will be sent.
u16ClusterId	Identifier of the cluster 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 23.1.4) containing the address of the remote node to which the request will be sent
pu8TransactionSequenceNumber	Pointer to a location to store the Transaction Sequence Number (TSN) of the request
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 profile: TRUE: Attributes are manufacturer-specific FALSE: Attributes are from ZigBee profile
u16ManufacturerCode	ZigBee Alliance code for the manufacturer that defined proprietary attributes (set to zero if attributes are from the ZigBee-defined profile - that is, if *bIsManufacturerSpecific* is set to FALSE)
psAttributeReportingConfigurationRecord	Pointer to array of structures, where each structure indicates the required configuration data for a single attribute (see Section 23.1.7)

Returns

E_ZCL_SUCCESS

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 (to a client) for all attributes on the server (regardless of whether automatic reporting has been configured on the attributes).

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.2.1).

After this function has been called and before the attribute report is sent, the event E_ZCL_CBET_REPORT_REQUEST is automatically 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 23.1.4) containing the address of the remote node to which the attribute report will be 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 will be sent
u8DestEndPoint	Number of endpoint on target client to which attribute report will be sent
hAPduInst	Handle of APDU instance that will contain the attribute report

Returns

E_ZCL_SUCCESS

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 JenOS Persistent Data Manager (PDM). 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 JenOS PDM. The function should only be called after the ZCL has been initialised. Following this function call, automatic attribute reporting can resume for the relevant attribute (e.g. following a power loss or device reset).

The 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).

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 profile: TRUE: Attribute is manufacturer-specific FALSE: Attribute is from ZigBee profile
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 23.1.5) containing the reporting configuration data for the attribute

Returns

E_ZCL_SUCCESS

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_ACF_RP bit) for an attribute. Setting this flag will allow automatic reporting to be configured and implemented for the attribute.

Note: It is not necessary to set this flag for attribute reports generated through calls to **eZCL_ReportAllAttributes()**, since the flag only affects the processing of ‘configure reporting’ commands.

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 profile: TRUE: Attribute is manufacturer-specific FALSE: Attribute is from ZigBee profile
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

eZCL_ReadAllAttributes

```
teZCL_Status eZCL_ReadAllAttributes(
                    uint8 u8SourceEndPointId,
                    uint8 u8DestinationEndPointId,
                    uint16 u16ClusterId,
                    bool bDirectionIsServerToClient,
                    tsZCL_Address *psDestinationAddress,,
                    uint8 *pu8TransactionSequenceNumber,
                    bool bIsManufacturerSpecific,
                    uint16 u16ManufacturerCode);
```

Description

This function can be used to send a 'read attributes' request to a cluster on a remote endpoint, in order to read either all client attributes or all server attributes, depending on the type of cluster instance (client or server). Note that read access to cluster attributes on the remote node must be enabled at compile-time as described in Section 1.2.

You must specify the endpoint on the local node from which the request is to be sent. The obtained attribute values will be written to the shared structure on this endpoint.

You must 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 that when sending requests to multiple endpoints through a single call to this function, multiple responses will subsequently be received from the remote endpoints.

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

You must specify the manufacturer code if the cluster is manufacturer-specific.

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. Once all received attribute values have been parsed, the event E_ZCL_CBET_READ_ATTRIBUTES_RESPONSE is generated.

The response may not contain values for all requested attributes and so further responses may follow. The first E_ZCL_CBET_READ_ATTRIBUTES_RESPONSE should prompt the application to call **eZCL_HandleReadAttributesResponse()** in order to ensure that all cluster attributes are received from the remote endpoint.

Parameters

u8SourceEndPointId	Number of the local endpoint through which the request will be sent
u8DestinationEndPointId	Number of the remote endpoint to which the request will be sent. Note that this parameter is ignored when sending to address types 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 read: TRUE: Cluster server to client FALSE: Cluster client to server
psDestinationAddress	Pointer to a structure (see Section 23.1.4) containing the address of the remote node to which the request will be sent
pu8TransactionSequenceNumber	Pointer to a location to store the Transaction Sequence Number (TSN) of the request
bIsManufacturerSpecific	Indicates whether attributes are manufacturer-specific or as defined in relevant ZigBee profile: TRUE: Attributes are manufacturer-specific FALSE: Attributes are from ZigBee profile
u16ManufacturerCode	ZigBee Alliance code for the manufacturer that defined proprietary attributes (set to zero if attributes are from the ZigBee-defined profile - that is, if *bIsManufacturerSpecific* is set to FALSE)

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

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).

The function should be called following a call to **eZCL_ReadAllAttributes()**. **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. Any further attribute values obtained will be written to the local shared device structure containing the attributes.

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

Parameters

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

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 profile: TRUE: Attribute is manufacturer-specific FALSE: Attribute is from ZigBee profile
blsClientAttribute	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

eZCL_WriteLocalAttributeValue

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

Description

This function can be used to write 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 will be 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 profile: TRUE: Attribute is manufacturer-specific FALSE: Attribute is from ZigBee profile
bIsClientAttribute	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

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 (e.g. **eSE_RegisterIPDEndPoint()** for an IPD) 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

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, you may want to use this function to disable application-level security on the local device so that the device will send all future communications with only network-level security. This is useful when transmitted packets need to be easily accessed, e.g. during over-air tests performed using a packet sniffer.

Parameters

eSecuritySupported	Required 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

23. 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.

23.1 General Structures

23.1.1 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

23.1.2 tsZCL_ClusterDefinition

This structure defines a cluster used on a device:

```
typedef struct
{
    uint16                      u16ClusterEnum;
    bool_t                      bIsManufacturerSpecificCluster;
    teZCL_ZCLSendSecurity       eSendSecurity;
    uint16                      u16NumberOfAttributes;
    tsZCL_AttributeDefinition  *psAttributeDefinition;
}  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
- `eSendSecurity` indicates the type of security used in cluster communications, one of:
  - E_ZCL_SECURITY_NETWORK - network-level security
  - E_ZCL_SECURITY_APPLINK - application-level security
- `u16NumberOfAttributes` indicates the number of attributes in the cluster
- `psAttributeDefinition` is a pointer to an array of attribute definition structures - see Section 23.1.3

23.1.3 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 bitmap of flags relating to the attribute
- `eAttributeDataType` is the data type of the attribute - see Section 24.1.3
- `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

23.1.4 tsZCL_Address

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

```
typedef struct PACK
{
  eSE_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 24.1.1)
- `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 24.1.2)

23.1.5 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 will be received or sent:
  - 0x00: Configures how attribute reports will be sent by the server - the following fields are included in the message payload: `eAttributeDataType`, `u16MinimumReportingInterval`, `u16MaximumReportingInterval`, `uAttributeReportableChange`
  - 0x01: Configures how attribute reports will be 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 will cause 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.

23.1.6 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 24.1.4)
- `sAttributeReportingConfigurationRecord` is a configuration record structure (see Section 23.1.5), but only the fields `u16AttributeEnum` and `u8DirectionIsReceived` are used in the response

23.1.7 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 will be received or sent
  - 0x00: Specifies that required information details how a report will be sent by the server
  - 0x01: Specifies that required information details how a report will be received by the client
- `u16AttributeEnum` is the identifier of the attribute to which the required reporting configuration information relates

23.1.8 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 24.1.3)
- `eAttributeStatus` is the status of the read operation (0x00 for success or an error code - see Section 24.1.4 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 23.2, when the field `eEventType` is set to E_ZCL_CBET_READ_INDIVIDUAL_ATTRIBUTE_RESPONSE.

23.1.9 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 23.2, when the field `eEventType` is set to E_ZCL_CBET_DEFAULT_RESPONSE.

23.1.10 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 (see Section).

```
  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 24.1.3)

The above structure is contained in the `tsZCL_CallBackEvent` event structure, detailed in Section 23.2, when the field `eEventType` is set to E_ZCL_CBET_DISCOVER_INDIVIDUAL_ATTRIBUTE_RESPONSE.

23.1.11 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 24.1.5)

23.1.12 tsZCL_OctetString

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

Octet Count, N (1 octet)	**Data** (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.

23.1.13 tsZCL_CharacterString

This structure contains information on a ZCL character string. This string is of the 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, e.g. 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.

23.1.14 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

23.1.15 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 23.1.2
- `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 initialised 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)

23.2 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;
#ifdef EZ_MODE_COMMISSIONING
       tsZCL_EZModeBindDetails                sEZBindDetails;
       tsZCL_EZModeGroupDetails               sEZGroupDetails;
#endif
   }uMessage ;
   ZPS_tsAfEvent                             *pZPSevent;
   tsZCL_ClusterInstance                     *psClusterInstance;
} tsZCL_CallBackEvent;
```

where

- `eEventType` specifies the type of event generated - see Section 24.3
- `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 24.2
- `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 23.1.8
  - `sDefaultResponse` contains the response to a request (other than a read request) - see Section 23.1.9
  - `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 23.1.14

- `sAttributeReportingConfigurationRecord` contains the attribute reporting configuration data from the 'configure reporting' request for an attribute - see Section 23.1.5
- `sAttributeReportingConfigurationResponse` is reserved for future use
- `sAttributeDiscoveryResponse` contains the details of an attribute reported in a 'discover attributes' response - see Section 23.1.10
- `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
- `sEZBindDetails` is only available if the EZ-mode Commissioning module is enabled (EZ_MODE_COMMISSIONING is TRUE) and contains details of a binding made with a cluster on a remote endpoint - see Section 21.7.1
- `sEZGroupDetails` is only available if the EZ-mode Commissioning module is enabled (EZ_MODE_COMMISSIONING is TRUE) and contains details of the addition of a remote endpoint to a group - see Section 21.7.2

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) which caused the ZCL event
- `psClusterInstance` is a pointer to the cluster instance structure which holds the information relating to the cluster being accessed

24. 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.

24.1 General Enumerations

24.1.1 Addressing Modes (eZCL_AddressMode)

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

```
typedef enum PACK
{
    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_ENUM_END,
} teZCL_AddressMode;
```

The above enumerations are described in the table below.

Enumeration	Description
E_ZCL_AM_BOUND	Use one or more bound nodes/endpoints
E_ZCL_AM_GROUP	Use a pre-defined group address
E_ZCL_AM_SHORT	Use a 16-bit network address
E_ZCL_AM_IEEE	Use a 64-bit IEEE/MAC address
E_ZCL_AM_BROADCAST	A broadcast (see Section 24.1.2)
E_ZCL_AM_NO_TRANSMIT	Do not transmit

Table 24: Addressing Mode Enumerations

The required addressing mode is specified in the structure `tsZCL_Address` (see Section 23.1.4).

24.1.2 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 24.1.1)):

```
typedef enum PACK
{
    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.

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 Co-ordinator and Routers

Table 25: Broadcast Mode Enumerations

The required broadcast mode is specified in the structure `tsZCL_Address` (see Section 23.1.4).

24.1.3 Attribute Types (teZCL_ZCLAttributeType)

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

```
typedef enum PACK
{
    /* 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;
```

24.1.4 Command Status (teZCL_CommandStatus)

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

```
typedef enum PACK
{
    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_HARDWARE_FAILURE =0xc0,
    E_ZCL_CMDS_SOFTWARE_FAILURE,
    E_ZCL_CMDS_CALIBRATION_ERROR
} teZCL_CommandStatus;
```

The above enumerations are described in the table below.

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	A command that acts across all profiles 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 supported or has unknown manufacturer code
E_ZCL_CMDS_UNSUP_MANUF_GENERAL_COMMAND	Manufacturer-specific ZCL command is not supported 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
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

Table 26: Command Status Enumerations

Enumeration	Description
E_ZCL_CMDS_CALIBRATION_ERROR	Error occurred during calibration

Table 26: Command Status Enumerations

24.1.5 Report Attribute Status (teZCL_ReportAttributeStatus)

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

```
typedef enum PACK
{
    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.

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

Table 27: Report Attribute Status Enumerations

24.1.6 Security Level (teZCL_ZCLSendSecurity)

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

```
typedef enum PACK
{
    E_ZCL_SECURITY_NETWORK = 0x00,
    E_ZCL_SECURITY_APPLINK,
    E_ZCL_SECURITY_ENUM_END
} teZCL_ZCLSendSecurity;
```

The above enumerations are described in the table below.

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

Table 28: Security Level Enumerations

24.2 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 PACK
{
   // 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_ENUM_END
    } teZCL_Status;
```

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
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)

Table 29: General Return Code Enumerations

Enumeration	Description
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_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_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. This return code is returned from SE function calls
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 synchronised by calling **vZCL_SetUTCTime()**. This error code is returned by functions that require time to be synchronised, e.g. **eSE_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
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

Table 29: General Return Code Enumerations

* 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

24.3 ZCL Event Enumerations

The ZCL event types are enumerated in the `teZCL_CallBackEventType` structure below and described in Table 30. An event must be wrapped in a structure of type `tsZCL_CallBackEvent`, detailed in Section 23.2, 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 22.1. Event handling is fully described in Chapter 3.

```
typedef enum PACK
{
    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_ENUM_END
} teZCL_CallBackEventType;
```

The above enumerations are described in the table below.

Event Type Enumeration	Description
E_ZCL_CBET_LOCK_MUTEX	Indicates that a mutex needs to be locked by the application
E_ZCL_CBET_UNLOCK_MUTEX	Indicates that a mutex needs to be unlocked by the application
E_ZCL_CBET_UNHANDLED_EVENT	Indicates that a stack event has been received that cannot be handled by the ZCL (e.g. a Data Confirm)
E_ZCL_CBET_READ_INDIVIDUAL_ATTRIBUTE_ RESPONSE	Generated for each attribute included in a ‘read attributes’ response (this event is often ignored by an SE application)
E_ZCL_CBET_READ_ATTRIBUTES_RESPONSE	Indicates that a ‘read attributes’ response has been received and that the local shared structure has been updated
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 cannot 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
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 will be allowed or disallowed

Table 30: ZCL Event Types

Event Type Enumeration	Description
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
E_ZCL_CBET_ZGP_DATA_IND_ERROR	Indicates that a ZigBee Green Power data indication error has occurred

Table 30: ZCL Event Types

Note: The structure `teZCL_CallBackEventType` is extended by the EZ-mode Commissioning module with the events listed and described in Section 21.4. These events are only included if this module is used, in which case they are added after E_ZCL_CBET_ENUM_END.

Part IV: Appendices

A. Mutex Callbacks

The mutexes provided by JenOS (Jennic Operating System) are designed such that a call to **OS_eEnterCriticalSection()** must be followed by a call to **OS_eExitCriticalSection()**, and must not be followed by another call to **OS_eEnterCriticalSection()**, i.e. 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 **OS_eEnterCriticalSection()**, so that it also participates in the counting mutex rather than directly taking the binary OS critical section. Similarly, the shared structure should be released using **vUnlockZCLMutex()**.

The code below uses a single OS resource for all endpoints and the general callback function. It defines a file scope counter that is the mutex count related to the OS 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)
    {
        OS_eEnterCriticalSection(mutexZCL);
    }
    u32ZCLMutexCount++;
}

void vUnlockZCLMutex(void)
{
    u32ZCLMutexCount--;
    if (u32ZCLMutexCount == 0)
    {
        OS_eExitCriticalSection(mutexZCL);
    }
}
```

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 device by sending a ‘configure reporting’ command to the server - see Appendix B.2. Remote devices can also query the attribute reporting configuration of the server - see Appendix B.5. Sending and receiving attribute reports are described in Appendix B.3 and Appendix B.4.

Attribute reporting is an optional feature and is not supported by all devices.

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.2.

B.2 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.2.1.

If attribute reports are to be prompted purely by the application then no further configuration is required. However, if automatic attribute reporting is to be implemented then the reports must be configured as described in Appendix B.2.2.

B.2.1 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. Options that are specific to Smart Energy (SE) are also listed.

Server Options

To enable a server to generate attribute reports according to configured reporting rules, add the following option:

```
ZCL_ATTRIBUTE_REPORTING_SERVER_SUPPORTED
```

Note: Attribute reporting does not need to be enabled with this macro if the reports will only be generated via function calls (e.g. when using Smart Energy mirroring).

To enable a server to handle ‘configure reporting’ commands and reply with ‘configure reporting’ responses, add the following option:

```
ZCL_CONFIGURE_ATTRIBUTE_REPORTING_SERVER_SUPPORTED
```

To enable a server to handle ‘read reporting configuration’ commands and reply with ‘read reporting configuration’ responses, add the following option:

```
ZCL_READ_ATTRIBUTE_REPORTING_CONFIGURATION_SERVER_SUPPORTED
```

Client Options

To enable a client to receive attribute reports from a server, add the following option:

```
ZCL_ATTRIBUTE_REPORTING_CLIENT_SUPPORTED
```

To enable a client to send ‘configure reporting’ commands and handle the ‘configure reporting’ responses, add the following option:

```
ZCL_CONFIGURE_ATTRIBUTE_REPORTING_CLIENT_SUPPORTED
```

To enable a client to send ‘read reporting configuration’ commands and handle the ‘read reporting configuration’ responses, add the following option:

```
ZCL_READ_ATTRIBUTE_REPORTING_CONFIGURATION_CLIENT_SUPPORTED
```

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.

SE-specific Options

For the Smart Energy (SE) profile, the following macros can be used to specify limits for automatic attribute reporting.

To limit the number of attributes for which automatic attribute reporting can be configured on a cluster, add the following option (where <n> is the maximum):

```
SE_NUMBER_OF_REPORTS <n>
```

To set a minimum time-interval between consecutive **triggered** attribute reports, add the following option (where <n> is the minimum time-interval, in seconds):

```
SE_SYSTEM_MIN_REPORT_INTERVAL <n>
```

To set the maximum time-interval between consecutive **periodic** attribute reports, add the following option (where <n> is the maximum time-interval, in seconds):

```
SE_SYSTEM_MAX_REPORT_INTERVAL <n>
```

Note: The application also sets limits on the time-intervals between consecutive attribute reports in periodic and triggered reporting (see Appendix B.2.2). These individual settings must not violate the above master values set at compile-time.

B.2.2 'Attribute Report Configuration' Commands

If automatic attribute reporting is to be employed between a cluster server and client, the reporting rules must be configured. These rules are profile-specific (refer to the appropriate ZigBee profile specification) but generally 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 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 2: Before automatic reporting can be configured on an attribute, the 'reportable flag' must be set for the attribute on the cluster server (if it is not pre-set in the profile) using the function **eZCL_SetReportableFlag()**. Also refer to Appendix B.7.

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 `tsZCL_AttributeReportingConfigurationRecord` structure must be specified which contains the details of the required configuration - this structure includes a pointer to an array of configuration records, one record per attribute for which reporting is to be configured (see Section 23.1.5).

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 23.2) for this event:

- The `uMessage` field contains a structure of the type `tsZCL_AttributeReportingConfigurationRecord` (see Section 23.1.5).
- 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.6.2).

Once attribute reporting has been configured 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 using the JenOS PDM, as described in Appendix B.6.

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 23.2) for this event, the `uMessage` field contains a structure of the type `tsZCL_AttributeReportingConfigurationResponse` (see Section 23.1.6). In this structure:

- The `eCommandStatus` field indicates the status of the attribute reporting configuration for the attribute.
- The `tsZCL_AttributeReportingConfigurationRecord` structure (Section 23.1.5) 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 will be 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.

B.3 Sending Attribute Reports

If automatic attribute reporting has been configured between the cluster server and a client (as described in Appendix B.2), the reporting of the relevant attributes will begin immediately after configuration. Attribute reports will be 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 attribute reports for all its attributes, when needed, by calling the function **eZCL_ReportAllAttributes()**. This function sends an attribute report containing the current attribute values to one or more clients specified in the function call. Use of this function 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.*

B.4 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.2.1).

When an attribute report is received from the server, the attribute values are written to the shared structure on the client and events are generated (in much the same way as for a 'read attributes' response) - the ZCL software performs the following steps:

1. Generates an E_ZCL_CBET_LOCK_MUTEX event for the relevant endpoint callback function, which should lock the mutex that protects the shared device structure on the client.
2. Writes the new attribute values to the shared device structure on the client.

3. Generates an E_ZCL_CBET_UNLOCK_MUTEX event for the endpoint callback function, which should now unlock the mutex that protects the shared device structure (other application tasks can now access the structure).
4. For each attribute 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.
5. 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 23.2) 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.

B.5 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 23.1.7).

Receiving a ‘Read Reporting Configuration’ Command (on Server)

The server will automatically process an incoming ‘read reporting configuration’ command without assistance from the application. Callback events are not generated. However, the server will generate 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 23.2) for this event, the `uMessage` field contains a structure of the type `tsZCL_AttributeReportingConfigurationResponse` (see Section 23.1.6) - this is the same structure as used in attribute reporting configuration, described in Appendix B.2.2.

In this structure:

- The `eCommandStatus` field indicates the status of the request.
- The `tsZCL_AttributeReportingConfigurationRecord` structure (see Section 23.1.5) 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.

B.6 Storing an Attribute Reporting Configuration

During the configuration of automatic attribute reporting, described in Appendix B.2.2, 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.

B.6.1 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 JN51xx 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.2.2. 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.6.2). Data storage in NVM can be performed under application control using the JenOS Persistent Data Manager (PDM), described in the *JenOS User Guide (JN-UG-3075)*.

On a 'cold start' of the JN51xx 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 initialised). To do this, the JenOS PDM 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.

B.6.2 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.2.1). 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 economise 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.

Minimised Data Storage

It may be possible to optimise 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 (e.g. a bitmap), so does not have
a reportable change
} tsZCL_PersistedAttributeReportingConfigurationRecord;
```

B.7 Profile Initialisation of Attribute Reporting

This section summarises the calls and definitions related to attribute reporting that are used within an application profile.

Note: The information in this section is only useful to developers who are creating their own application profiles.

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 by a profile using the **u32ZCL_Heap** macro - for example, in Smart Energy, we have:

PRIVATE uint32 u32ZCL_Heap[
ZCL_HEAP_SIZE(*SE_NUMBER_OF_ENDPOINTS***,**
*SE_NUMBER_OF_TIMERS***,**
*SE_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, in Smart Energy, we have:

```
sConfig.u8NumberOfReports = SE_NUMBER_OF_REPORTS;
sConfig.u16SystemMinimumReportingInterval =
SE_SYSTEM_MIN_REPORT_INTERVAL;
sConfig.u16SystemMaximumReportingInterval =
SE_SYSTEM_MAX_REPORT_INTERVAL;
```

The default value for SE_NUMBER_OF_REPORTS is 10 but this can be over-ridden in the application's **zcl_options.h** file - see Appendix B.2.1.

A server that supports automatic attribute reporting should have the 'reportable flag' (E_ZCL_AF_RP configuration bit) set for any attributes that are reportable. If a server receives a 'configure reporting' command for an attribute that does not have this flag set, it will return an error and not allow the attribute to be reported. This bit setting is not required for attribute reports generated through calls to the function **eZCL_ReportAllAttributes()**, as the flag only affects the processing of a 'configure reporting' command.

Attribute definitions that are part of standard profiles, such as Home Automation and Smart Energy, will not normally have the reportable flag set. The application on the server should set this flag for those attributes on which reporting is to be permitted. This can be done using the function **eZCL_SetReportableFlag()**.

C. JN51xx Bootloaders

JN516x:

During start-up, the JN516x bootloader (provided in internal Flash memory) searches for a valid application image in internal Flash memory. If one is present then the device will boot directly from Flash memory. If no image is found then the bootloader will seach through an external Flash device for an image header.

An application image can be stored in any sector of external Flash memory, except the final sector (if it has been reserved for persistent data storage by the application). The bootloader searches through the Flash memory, looking at the start of each sector for the image header that identifies the current application image. If a valid header is detected then the image is loaded into internal Flash memory and executed.

JN5148-Z01:

During start-up, the JN5148-Z01 bootloader (provided in on-chip ROM) copies the 'current' application image from external Flash memory to on-chip RAM and executes the application.

An application image can be stored in any sector of external Flash memory, except the final sector (which is reserved for persistent data storage). The bootloader searches through the Flash memory, looking at the start of each sector for the image header that identifies the current application image. If a valid header is detected then the image is loaded into RAM and executed. This is called a multi-image bootloader.

D. OTA Extension for Dual-Processor Nodes

This appendix describes use of the Over-the-Air (OTA) Upgrade cluster (introduced in Chapter 20) for a ZigBee PRO network consisting of dual-processor nodes that each contain a JN51xx wireless microcontroller and a co-processor.

The co-processor is connected to the JN51xx device via a serial interface and may have its own external storage device, as depicted in Figure 7 below.

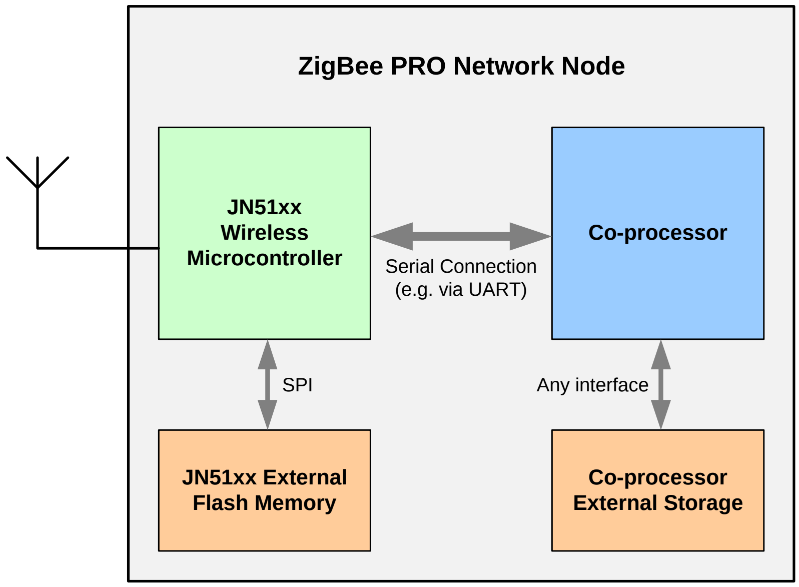

ZigBee PRO Network Node
JN51xx Wireless Microcontroller
Co-processor
Serial Connection (e.g. via UART)
SPI
Any interface
JN51xx External Flash Memory
Co-processor External Storage

Figure 7: 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 JN51xx 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. In the case of a Smart Energy network, this node will be the ESP which receives software updates from the utility company via the backhaul network.
2. The co-processor on the OTA server node either saves the received software image in its own storage device or (normally) passes the image to the JN51xx microcontroller for storage in its external Flash memory device.
3. The OTA Upgrade cluster server running on the JN51xx device distributes the software update over-the-air to the appropriate ZigBee PRO network nodes, as described in Section 20.3.
4. On a target node, the OTA Upgrade cluster client running on the JN51xx 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 JN51xx device or the co-processor. In a Smart Energy network, this node may typically be an IPD or a Metering Device.

5. The OTA Upgrade cluster client running on the JN51xx 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 8 below for the case of a Smart Energy network in which the co-processor application on an IPD is updated via an OTA upgrade and the image is stored in the target co-processor's own storage device.

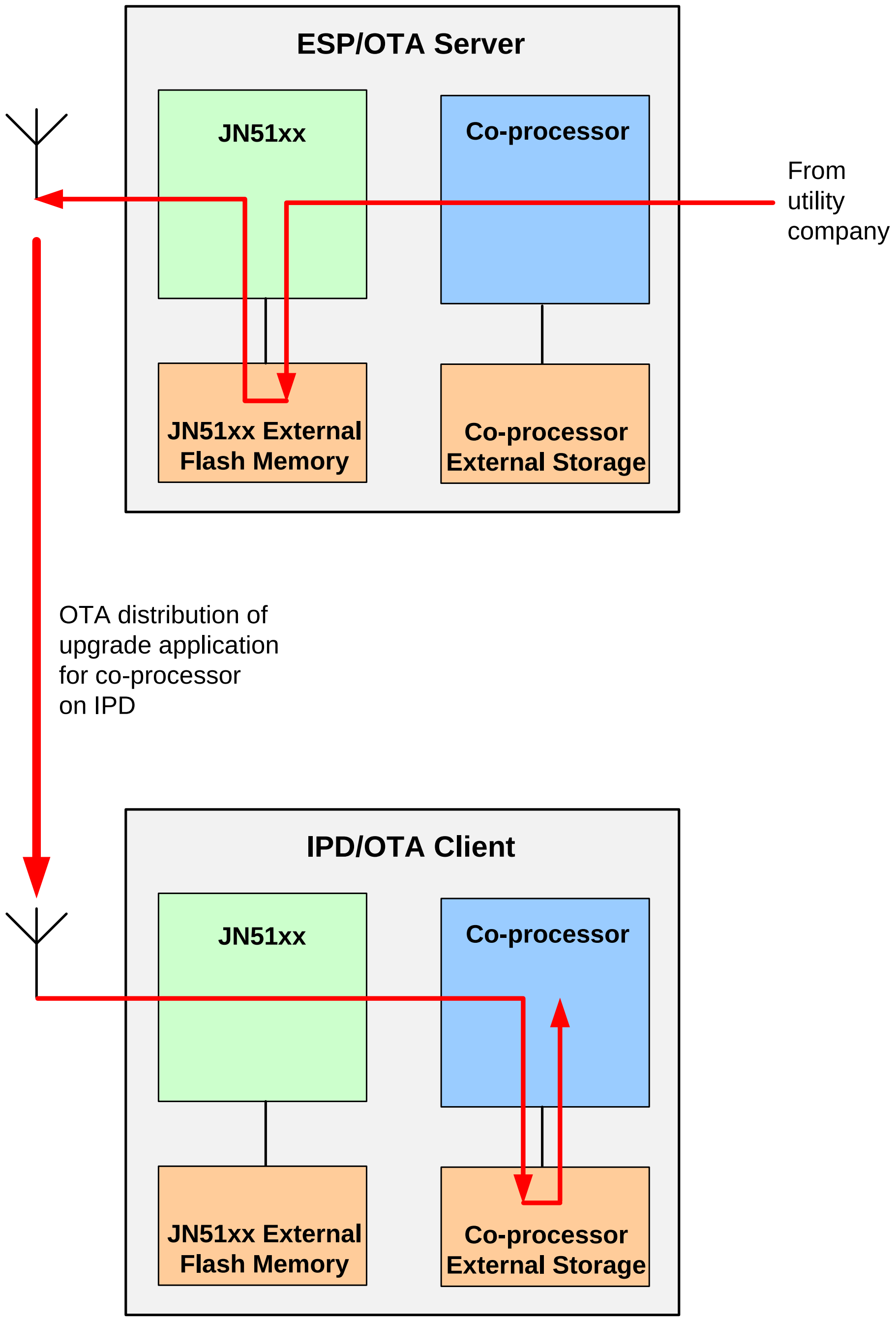

ESP/OTA Server
JN51xx
Co-processor
From utility company
JN51xx External Flash Memory
Co-processor External Storage
OTA distribution of upgrade application for co-processor on IPD
IPD/OTA Client
JN51xx
Co-processor
JN51xx External Flash Memory
Co-processor External Storage

Figure 8: Example of OTA Upgrade of Co-processor Application on IPD

D.1 Application Upgrades for Different Target Processors

In a ZigBee PRO network containing dual-processor nodes (with a JN51xx microcontroller and a co-processor), an application upgrade can be targeted at any of the following processors:

- OTA server node processors:
  - JN51xx microcontroller
  - Co-processor
- OTA client node processors:
  - JN51xx microcontroller
  - 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.

Target Processor for Application Upgrade	**Intermediate Processors during Application Upgrade**			
	OTA Server		**OTA Client**	
	Co-processor	**JN51xx**	**JN51xx**	**Co-processor**
OTA Server Co-processor	Co-processor saves new image to its external storage and performs update	-	-	-
OTA Server JN51xx	Co-processor passes new image to server JN51xx device *	JN51xx saves image to Flash memory and performs update *	-	-
OTA Client JN51xx	Co-processor passes new image to server JN51xx device *	JN51xx saves image to Flash memory and then sends it over-the-air to client *	JN51xx receives image, saves it to Flash memory and performs update	-
OTA Client Co-processor	Co-processor passes new image to server JN51xx device *	JN51xx saves image to Flash memory and then sends it over-the-air to client *	JN51xx receives image and saves it to Flash memory or to co-processor storage device	Co-processor performs update

Table 31: Processor Roles in Application Upgrade

* If insufficient space in Flash memory, image may be stored in co-processor storage - see Appendix D.3

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 D.2.

D.2 Application Upgrade Scenarios

In the application upgrade scenarios described in this section, a new software image is:

1. received from an external source by the co-processor in the OTA server node (e.g. in the case of a Smart Energy network, the software is received by the ESP via the backhaul network from the utility company)
2. passed from the co-processor via a serial connection to the JN51xx microcontroller in the OTA server node (see Note 1 below)
3. saved by the JN51xx device to its external Flash memory

Once saved to Flash memory, the fate of the new software image depends on which processor is to have its application updated - JN51xx device in the OTA server, JN51xx device in an OTA client or the co-processor in an OTA client. If the target processor is in an OTA client, the server must transmit the image over-the-air.

Note 1: If the Flash memory of the JN51xx device has insufficient free space to store a new software image, the image may be saved to the external storage device of the co-processor. The JN51xx application must make the decision of where the image will be stored. Refer to Appendix D.3 for more details of this scenario.

Note 2: This section does not describe the case of the co-processor on the OTA server node updating its own application, as this upgrade mechanism is specific to the co-processor.

Note 3: The OTA functions referenced in this section are fully detailed in Section 20.9.

The OTA server may need to store different upgrade images for different nodes (possibly from different manufacturers). The maximum number of such images that can be stored must be specified as a compile-time option in the **zcl_options.h** file by defining the values of:

- OTA_MAX_IMAGES_PER_ENDPOINT which represents the maximum number of images that may be stored in JN51xx external Flash memory
- OTA_MAX_CO_PROCESSOR_IMAGES which represents the maximum number of images that may be stored in co-processor external storage

The upgrade images stored on the server are indexed from zero, with the Flash memory images numbered first - for further details, refer to Appendix D.4.

Flash memory sectors are allocated to upgrade images using the OTA function **eOTA_AllocateEndpointOTASpace()**. This function takes as input the maximum number of images to be stored in Flash memory and the number of sectors to be allocated per image. The start sectors for the images must also be specified in an array, where the array index identifies the image (see Appendix D.4). The JN51xx application is responsible for deciding which index value and therefore which Flash sectors are allocated to a new upgrade image.

When a new software image is acquired by the co-processor on the OTA server node (e.g. from the utility company) and this image is to be passed to the JN51xx device for storage in its external Flash memory, the co-processor application must prompt the JN51xx application to perform this storage. The co-processor application must send custom messages via the serial interface to the JN51xx application in order to request certain OTA function calls, as follows:

1. The Flash memory sectors that will be used to store the new image must first be erased by specifying the relevant image index in a call to the function **eOTA_EraseFlashSectorsForNewImage()**.
2. If the new image is a client image, the current equivalent image in Flash memory should now be invalidated using the function **eOTA_InvalidateStoredImage()**.
3. On receiving each block of the new image from the co-processor, the function **eOTA_FlashWriteNewImageBlock()** must be called to write the block to the relevant sector of Flash memory.
4. After receiving the final block of the new image, the co-processor will indicate the end of the image and the next function call depends on whether the image is destined for the server itself or for one or more clients. The required function calls are specified in the subsections below.

The above process is illustrated in Figure 9 below.

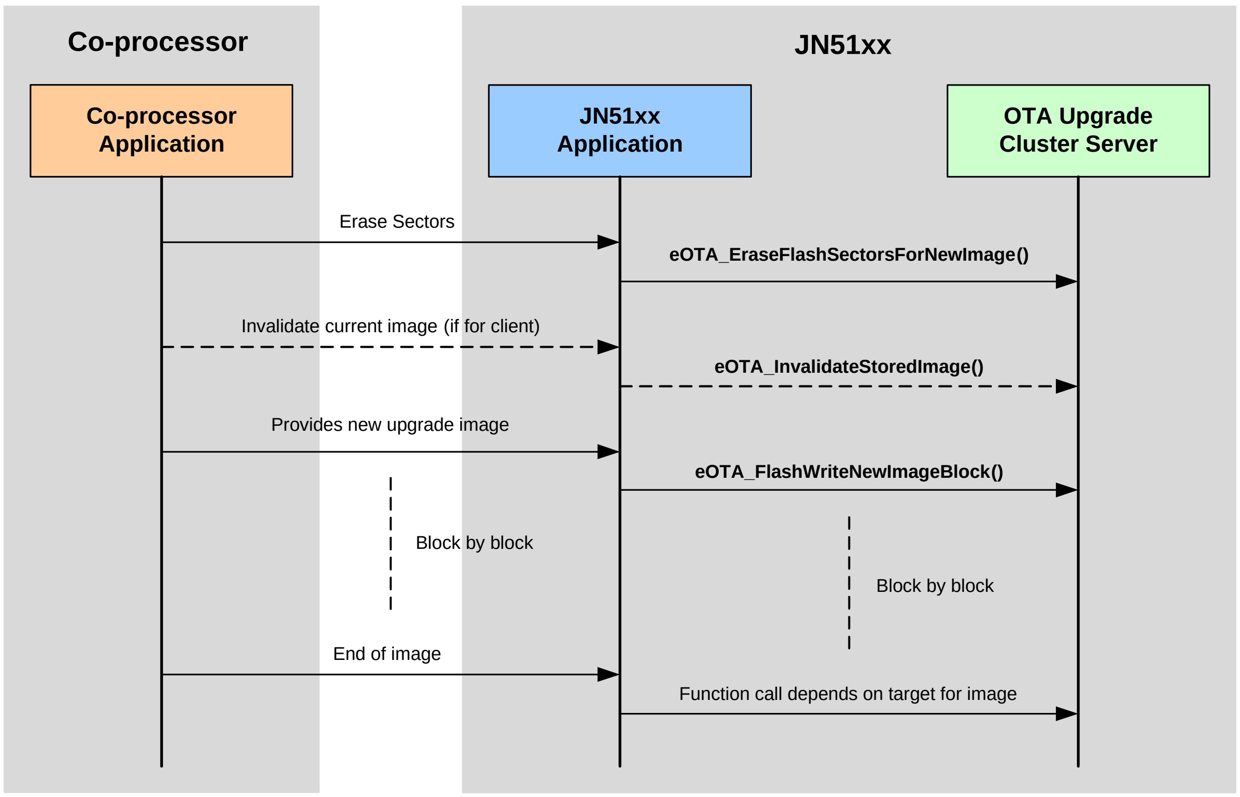

Co-processor
JN51xx
Co-processor Application
JN51xx Application
OTA Upgrade Cluster Server
Erase Sectors
eOTA_EraseFlashSectorsForNewImage()
Invalidate current image (if for client)
eOTA_InvalidateStoredImage()
Provides new upgrade image
eOTA_FlashWriteNewImageBlock()
Block by block
Block by block
End of image
Function call depends on target for image

Figure 9: Saving a New Upgrade Image to Flash Memory on Server

Once the new upgrade image is available in Flash memory on the OTA server node, it can be distibuted by the server according to which processor(s) it is intended for:

- JN51xx device in the OTA server - see Appendix D.2.1
- JN51xx device in one or more OTA clients - see Appendix D.2.2
- Co-processor in one or more OTA clients - see Appendix D.2.3

D.2.1 Loading Image into JN51xx in OTA Server Node

This section describes how an application image which is destined for the JN51xx device on the OTA server node is loaded into internal Flash memory or RAM on the device and run. It is assumed that the image has been saved to the external Flash memory of the JN51xx device, as illustrated in Figure 9.

Once all the image blocks have been transferred into Flash memory and the end of the image has been signalled by the co-processor, the JN51xx application must call the function **eOTA_ServerSwitchToNewImage()**. This function will reset the JN51xx device and cause the device to boot from the new image in Flash memory, as described in the last two steps of the upgrade process detailed in Section 20.6. Thus, the JN51xx device will now be running the upgrade application.

The old application image in Flash memory is no longer needed and its sectors can now be re-used to store another upgrade image for the server or clients. The old image must first be invalidated using the function **eOTA_InvalidateStoredImage()**.

D.2.2 Distributing Image to JN51xx in OTA Client Node(s)

This section describes how an application image which is destined for the JN51xx device on an OTA client node is downloaded from the OTA Upgrade server and run on the target JN51xx device. It is assumed that the image has been saved to the external Flash memory of the JN51xx device on the OTA server node, as illustrated in Figure 9.

Once all the image blocks have been transferred into Flash memory on the OTA server node and the end of the image has been signalled by the co-processor, the OTA Upgrade server must advertise the new client image so that clients can request the new image to be downloaded, save it to local Flash memory and then reboot the JN51xx device from this image - this process is exactly as described in Section 20.6.

Note 1: The JN51xx device on an OTA client node must also be able to identify upgrade images that are destined for the co-processor. This identification is performed using image header information that is registered at node initialisation - see Appendix D.2.3.

Note 2: The maximum number of images that can be stored on the OTA client node must be defined in the **zcl_options.h** file, as described in the compile-time options in Section 20.12 (also refer to Appendix D.4).

D.2.3 Distributing Image to Co-processor in OTA Client Node(s)

This section describes how an application image which is destined for the co-processor on an OTA client node is downloaded from the OTA Upgrade server and run on the target device. It is assumed that the image has been saved to the external Flash memory of the JN51xx device on the OTA server node, as illustrated in Figure 9.

Once all the image blocks have been transferred into Flash memory on the OTA server node and the end of the image has been signalled by the co-processor, the new upgrade image can be distributed to the relevant OTA client nodes as follows:

Note 1: On an OTA client node, the image may be stored in the external Flash memory of the JN51xx device or in the external storage device of the co-processor - the storage device used is determined by the application. Both possibilities are covered in the process below.

Note 2: The maximum number of images that can be stored on the OTA client node must be defined in the **zcl_options.h** file, as described in the compile-time options in Section 20.12 (also refer to Appendix D.4).

1. The new upgrade image is advertised to a client as described in Steps 1 to 3 in Section 20.6

2. On receiving the Query Next Image Response from the server, the OTA Upgrade cluster client analyses the image details contained in the response, from which it determines whether the image is relevant to either the JN51xx device or the co-processor in the node.

   This assessment is performed using image header information that has been registered with the OTA Upgrade cluster client. During initialisation of the OTA client node, the co-processor application must notify the JN51xx application of the header information for the co-processor application image(s). The JN51xx application must then register this information with the OTA Upgrade cluster client by calling the function **eOTA_UpdateCoProcessorOTAHeader()**.

   An upgrade image for the co-processor can be stored in the external Flash memory of the JN51xx device or in the external storage device of the co-processor. It is the responsibility of an application (JN51xx or co-processor) to store an image in its own external storage device. In order to store an image in its associated Flash memory, the JN51xx application needs the image index and start sector for the Flash memory space where the image is to be stored. It can obtain this information from the `u8NextFreeImageLocation` and `u8ImageStartSector` fields of the `tsOTA_CallBackMessage` structure (see Section 20.10.21) in the Query Next Image Response event.

3. If the new image is destined for the co-processor, the OTA Upgrade cluster client will automatically request the upgrade image one block at a time by sending Image Block Requests to the server.

   On arrival at the server, an Image Block Request message triggers an Image Block Request event.

4. The server automatically responds to each block request with an Image Block Response containing a block of image data.

   After each image block received, the cluster client generates the event E_CLD_OTA_INTERNAL_COMMAND_CO_PROCESSOR_BLOCK_RESPONSE. The client uses this event to confirm that the received block is part of the image being downloaded for the co-processor. If this is the case, the JN51xx application must do one of the following, depending on where the image is being stored:

   - Pass the image block to the co-processor application for storage in the co-processor's own storage device
   - Call Flash memory access (read, write and erase) functions to save the image block to the relevant place in JN51xx Flash memory

Note: To perform Flash memory access operations, the JN51xx application can call user-defined functions (if any) provided through **vOTA_FlashInit()** (see Section 20.5) or Integrated Peripherals API functions, such as **bAHI_FullFlashProgram()** and **bAHI_FullFlashRead()** - for an example, refer to Appendix F.2. The start address in Flash memory for each image block must be tracked by the application.

5. 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 all the image blocks have been received:

   a) An E_CLD_OTA_INTERNAL_COMMAND_CO_PROCESSOR_IMAGE_DL_COMPLETE is generated by the client to indicate that the image transfer is complete.

   b) The image can optionally be verified - if saved in JN51xx Flash memory then it can be verified using the function **eOTA_VerifyImage()**, but if saved in the co-processor storage device then the co-processor must be requested to perform the verification.

   c) The 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_CoProcessorUpgradeEndRequest()** - if the image was saved to the co-processor storage device then this call must be prompted by the co-processor application. On arrival at the server, the Upgrade End Request message triggers an Upgrade End Request event.

6. 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.

7. The 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_CO_PROCESSOR_SWITCH_TO_NEW_IMAGE.

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.

8. Finally, it is the responsibility of the co-processor application to update itself with the new image. This upgrade mechanism is specific to the co-processor.

Steps 4-7 are illustrated below in Figure 10 for the case of saving to the JN51xx Flash memory device and in Figure 11 for the case of saving to the co-processor storage device.

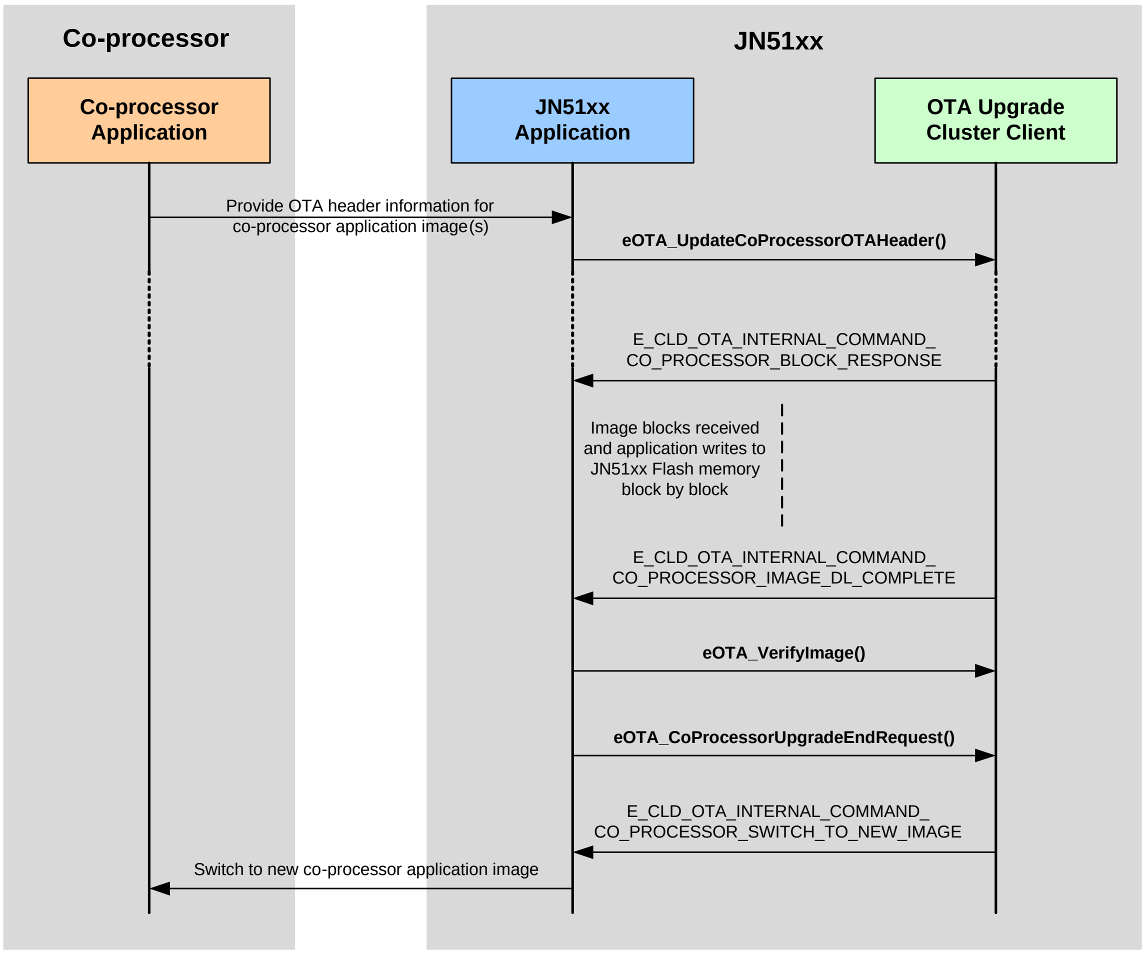

Co-processor
JN51xx
Co-processor Application
JN51xx Application
OTA Upgrade Cluster Client
Provide OTA header information for co-processor application image(s)
eOTA_UpdateCoProcessorOTAHeader()
E_CLD_OTA_INTERNAL_COMMAND_ CO_PROCESSOR_BLOCK_RESPONSE
Image blocks received and application writes to JN51xx Flash memory block by block
E_CLD_OTA_INTERNAL_COMMAND_ CO_PROCESSOR_IMAGE_DL_COMPLETE
eOTA_VerifyImage()
eOTA_CoProcessorUpgradeEndRequest()
E_CLD_OTA_INTERNAL_COMMAND_ CO_PROCESSOR_SWITCH_TO_NEW_IMAGE
Switch to new co-processor application image

Figure 10: Downloading Co-processor Image to JN51xx Flash Memory

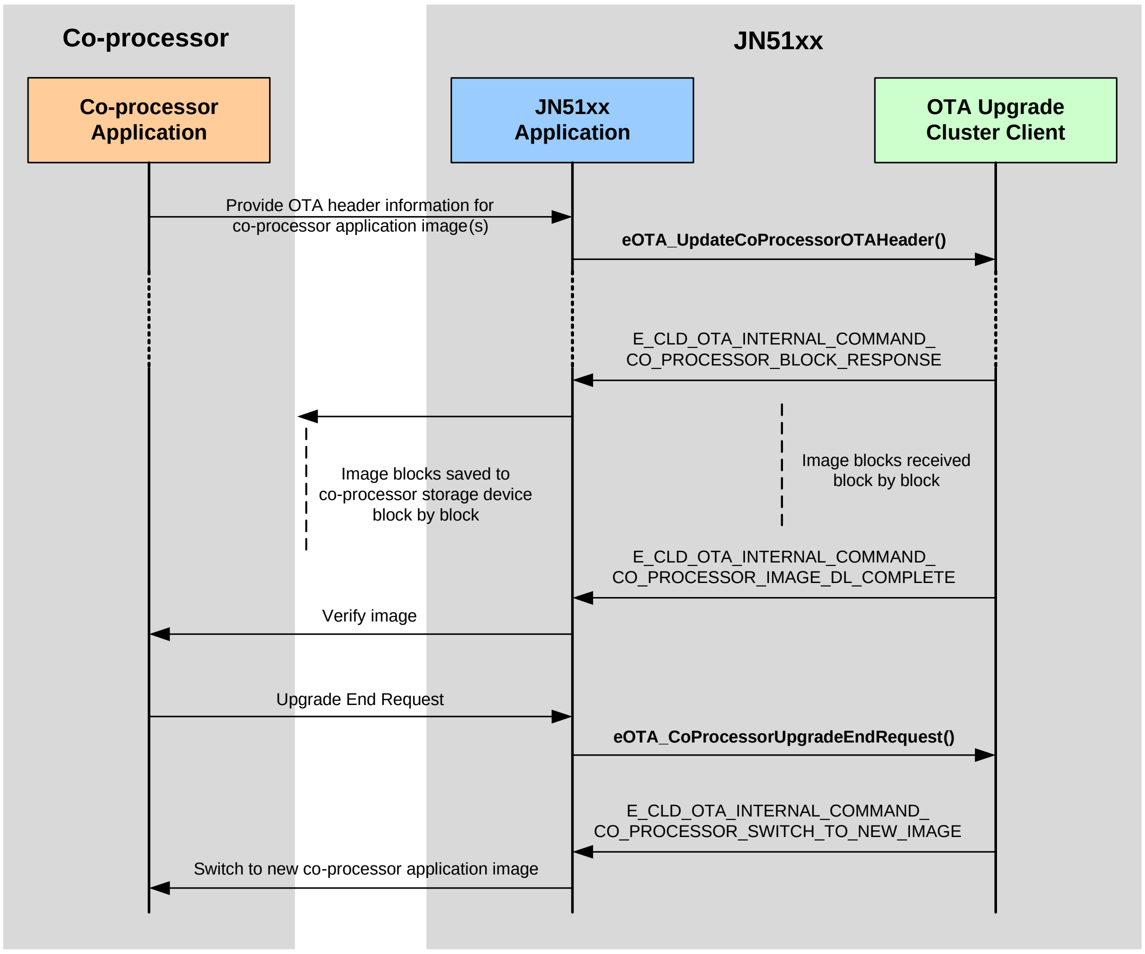

Co-processor
JN51xx
Co-processor Application
JN51xx Application
OTA Upgrade Cluster Client
Provide OTA header information for co-processor application image(s)
eOTA_UpdateCoProcessorOTAHeader()
E_CLD_OTA_INTERNAL_COMMAND_ CO_PROCESSOR_BLOCK_RESPONSE
Image blocks saved to co-processor storage device block by block
Image blocks received block by block
E_CLD_OTA_INTERNAL_COMMAND_ CO_PROCESSOR_IMAGE_DL_COMPLETE
Verify image
Upgrade End Request
eOTA_CoProcessorUpgradeEndRequest()
E_CLD_OTA_INTERNAL_COMMAND_ CO_PROCESSOR_SWITCH_TO_NEW_IMAGE
Switch to new co-processor application image

Figure 11: Downloading Co-processor Image to Own Storage Device

D.3 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 JN51xx 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 JN51xx application of the arrival of a new image, the JN51xx application must check whether there is sufficient Flash memory space for the image.
- If there is insufficient Flash memory space, the JN51xx 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 JN51xx 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 JN51xx device. After requesting and receiving the required image block from the co-processor, the JN51xx application must send the block to the relevant client by calling the function **eOTA_ServerImageBlockResponse()** to issue an Image Block Response.

D.4 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 JN51xx device.

Note: In the case of JN51xx 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 JN51xx 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 should be used is either 2 or 1, for JN5148 and JN516x respectively, to accommodate the currently active image.

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

and the image indices can take values in the range:

0 to (OTA_MAX_IMAGES_PER_ENDPOINT + OTA_MAX_CO_PROCESSOR_IMAGES - 1)

In fact, the indices of the images stored in JN51xx external Flash memory still take values in the range:

0 to (OTA_MAX_IMAGES_PER_ENDPOINT - 1)

while 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)

D.5 Multiple OTA Download Files

This section describes how multiple OTA files can be downloaded into a single device, where these files can be either dependent on or independent of each other.

D.5.1 Multiple Independent OTA Files

This section describes how multiple independent OTA files can be downloaded, e.g. when a co-processor is connected to the JN51xx and the image upgrades are independent of each other. This configuration must be specified when registering the co-processor OTA header, by calling the **eOTA_UpdateCoProcessorOTAHeader()** function with the *bIsCoProcessorImageUpgradeDependent* parameter set to FALSE.

On receiving an Image Notify command, the OTA client will send a Query Next Image Request command for both its own upgrade image and for any relevant co-processor images. If it receives a Query Next Image Response with status of SUCCESS for any one image then it will start a download of that image. If this is a JN51xx image then the client will follow the steps detailed in Section 20.6. If it is a co-processor image then the client will follow the steps in Appendix D.2.3. On completion of a download, the client will return to its normal state.

D.5.2 Multiple Dependent OTA Files

This section describes how multiple dependent OTA files can be downloaded, e.g. when a co-processor is connected to the JN51xx and the image upgrades are dependent on each other. This configuration must be specified when registering the co-processor OTA header, by calling the **eOTA_UpdateCoProcessorOTAHeader()** function with the *bIsCoProcessorImageUpgradeDependent* parameter set to TRUE.

On receiving an Image Notify command, the OTA client will send a Query Next Image command for its own upgrade image first, process the download and save it in external Flash memory. On completion, it will send an Upgrade End Request command with a status of REQUIRE_MORE_IMAGE and will generate the callback event E_CLD_OTA_INTERNAL_COMMAND_REQUEST_QUERY_NEXT_IMAGES. On actioning this event, the application must send a Query Next Image command for the next image by calling the **eOTA_ClientQueryNextImageRequest()** function. The client will then download and save the image as per steps 4 and 5 of Appendix D.2.3.

Once all dependant images have been downloaded, the OTA client will send an Upgrade End Request command with a status of SUCCESS.

After receiving the Upgrade End Response command, the client will count down to the upgrade time (specified in the Upgrade End Response) and, upon reaching it, will generate the event E_CLD_OTA_INTERNAL_COMMAND_CO_PROCESSOR_SWITCH_TO_NEW_IMAGE. Finally, it is the responsibility of the application to update the JN51xx and co-processor images with the newly downloaded images.

In order to initiate an upgrade of the JN51xx device, the application should call the function **eOTA_ClientSwitchToNewImage()**.

E. EZ-mode Commissioning Actions and Terminology

In the Home Automation Specification 1.2, ZigBee recommend terminology to be used in describing EZ-mode commissioning in HA product documentation. The aim of these recommendations is to ensure consistency between products and manufacturers, which will in turn provide users with a uniform experience of HA products.

The recommended terminology describes a number of actions that may be performed on an HA device (note that an individual action may not be valid on all device types). The recommended phrases for the actions are listed below in Table 32 - a description of each action is provided. The phrases and corresponding descriptions are quoted directly from the ZigBee Home Automation Specification 1.2.

ZigBee Action	User Action (bold) and Description (italics)
Join Network	**Press the Network button.** *Go find and join the first available HA network.*
Form Network	**Press and hold the Network button.** *For devices that can start a network.*
Allow Others To Join Network	**Press the Network button.** *For routers and coordinators only. Allows you to add more nodes to an existing network. This must have a mandatory timeout of 60 seconds.*
Restore Factory Fresh Settings	**Press and hold the Reset button.** *Restore the device settings to fresh state (also performs leave).*
Pair Devices	**Press the Binding button.** *End Device Bind Request. Bind to any device you can find matching clusters on. This will toggle the bind each time you do it. The ZigBee coordinator does the pairing.* *Example: a user would like to pair two devices (for example, a switch and a light).* • *A button on each device is pressed and the “pairing” is done using the end device bind request.* • *It is required that the Coordinator include the “bind manager”/End device response. The Bind manager uses the ZDP bind/unbind request to create the source binding in the devices.* • *If a device does not contain buttons, a proprietary remote control could be used to initiate the same function by sending a datagram to the device (emulating a button press).*
Enable Identify Mode	**Press the Binding button followed by a press on the selected user button (EP) to set to Identify.** *Sets the device in Identify mode for 60 seconds. This is used for adding devices to a group or creating a scene.*

Table 32: Recommended Phrases for Commissioning Actions

If a device does not support an action, the action must be listed in the device’s documentation as “Not Supported”.

F. Example Code Fragments

This appendix contains various fragments of example code.

F.1 Code Fragment of Image Verification Task

The code fragment in this section relates to the OTA image verification task described in Section 20.7.9.

A low-priority image verification task (such as APP_ImageVerifyTask) can be created in the JenOS Configuration Editor with the lowest priority set and the Autostart option set to FALSE. The task should be connected to all mutexes.

```
tsOTA_CallBackMessage *psMessage =
(tsOTA_CallBackMessage*)psEvent->uMessage.sClusterCustomMessage.pvCustomData;
if(psMessage->eEventId ==
E_CLD_OTA_INTERNAL_COMMAND_OTA_START_IMAGE_VERIFICATION_IN_LOW_PRIORITY)
{
#ifdef OTA_ACCEPT_ONLY_SIGNED_IMAGES
            u8ImageLocation = psMessage->u8NextFreeImageLocation;
/* Invoke low priority task to verify image */
     OS_eActivateTask(APP_ImageVerifyTask)
#endif
}

'APP_ImageVerifyTask' can be written as below -
OS_TASK(APP_ImageVerifyTask)
{

DBG_vPrintf(TRACE_IPD_NODE, "In APP_ImageVerifyTask \n");

#ifdef OTA_ACCEPT_ONLY_SIGNED_IMAGES

teZCL_Status eStatus;
eStatus = eOTA_VerifyImage(IPD_BASE_LOCAL_EP,
   FALSE,
         u8ImageLocation, //image location
   FALSE);
if(E_ZCL_SUCCESS != eStatus)
{

DBG_vPrintf(TRACE_IPD_NODE, " eOTA_VerifyImage Failed %d\n",eStatus);
}

eStatus = eOTA_HandleImageVerification(IPD_BASE_LOCAL_EP,
                           s_sDevice.sEsp.u8OtaEndPoint,
                           eStatus);

if(E_ZCL_SUCCESS != eStatus)
{

DBG_vPrintf(TRACE_IPD_NODE, " eOTA_HandleImageVerificatione Failed %d, Dest
endpoint=%d \n",eStatus,s_sDevice.sEsp.u8OtaEndPoint );
}
else
{
DBG_vPrintf(TRACE_IPD_NODE, " eOTA_HandleImageVerificatione Success %d, Dest
endpoint=%d \n",eStatus,s_sDevice.sEsp.u8OtaEndPoint );
}
#endif
}
```

F.2 Code Fragment for Flash Memory Access

The code fragment in this section is concerned with writing an OTA co-processor image to the Flash memory associated with a JN51xx device, using the standard function **bAHI_FullFlashProgram()** of the Integrated Peripherals API, detailed in the *JN516x/JN514x Integrated Peripherals API User Guides (JN-UG-3087/JN-UG-3066)*. The code below relates to the description in Appendix D.2.3.

```
tsOTA_CallBackMessage * psOTAMessage =
(tsOTA_CallBackMessage*)psEvent->uMessage.sClusterCustomMessage.pvCustomData;
if(psOTAMessage ->eEventId ==
E_CLD_OTA_INTERNAL_COMMAND_CO_PROCESSOR_BLOCK_RESPONSE)
{
if(psOTAMessage->uMessage.sImageBlockResponsePayload.u8Status ==
E_ZCL_SUCCESS)
{
bool_t bWriteStatus;
uint32 u32FlashOffset;
uint8 i;
if(psOTAMessage-
>uMessage.sImageBlockResponsePayload.uMessage.sBlockPayloadSuccess.u32FileOffset ==
0)
{ /* Erase the Flash sectors before start to write */
for(i=0;i<psOTAMessage->u8MaxNumberOfSectors;i++)
{
bAHI_FlashEraseSector(psOTAMessage->u8ImageStartSector[psOTAMessage-
>u8NextFreeImageLocation]+i);
}
}
u32FlashOffset = (psOTAMessage->u8ImageStartSector[psOTAMessage-
>u8NextFreeImageLocation] *(64*1024)) ;
u32FlashOffset += psOTAMessage-
>uMessage.sImageBlockResponsePayload.uMessage.sBlockPayloadSuccess.u32FileOffset;
bWriteStatus = bAHI_FullFlashProgram(u32FlashOffset,
psOTAMessage-
>uMessage.sImageBlockResponsePayload.uMessage.sBlockPayloadSuccess.u8DataSize,
psOTAMessage-
>uMessage.sImageBlockResponsePayload.uMessage.sBlockPayloadSuccess.pu8Data);
if(bWriteStatus == FALSE)
{
DBG_vPrintf(TRACE_ZCL_TASK, "Event : OTA flash write fail\n");
}
}
}
```

In the case of a dependent multiple-file download `psOTAMessage->u8NextFreeImageLocation` cannot be used as an image location.

- A JN514x application can use any image location except 0 and 1, since these locations are used for the device's own images (active and upgrade images)
  - OTA_MAX_IMAGES_PER_ENDPOINT must be defined as 2+OTA_MAX_CO_PROCESSOR_IMAGES
- A JN516x application can use any image location except 0, since this location is used to store the JN516x upgrade image
  - OTA_MAX_IMAGES_PER_ENDPOINT must be defined as 1+OTA_MAX_CO_PROCESSOR_IMAGES

Revision History

Version	Date	Comments
1.0	11-May-2011	First release (for Smart Energy)
1.1	23-May-2012	Made minor updates/corrections and added: • Attribute discovery and reporting • OTA extension for dual-processor nodes • Bootloader differences between JN5148 variants • New attribute access functions • Bound transmission management feature
1.2	03-Sept-2012	Made minor updates/corrections and added: • Commissioning cluster • New OTA Upgrade cluster features (rate limiting, page requests, device-specific file downloads)
1.3	15-Jan-2013	• Manual re-organised (now one chapter per cluster) • The following clusters added for ZigBee Light Link: Identify, Groups, Scenes, On/Off, On/Off Switch Configuration, Level Control, Colour Control • Basic cluster updated for ZigBee Light Link • New OTA Upgrade cluster functions added • Some structure definitions updated
1.4	24-Jan-2013	Content for ZigBee Light Link application profile updated
1.5	20-Feb-2013	Added the Level Control cluster function eCLD_LevelControlCommandStopWithOnOffCommandSend() and made various modifications/corrections
1.6	18-Apr-2013	Made minor updates/corrections and added: • 'Cluster instance create' functions for custom endpoints • 'ZCL Functions' chapter containing functions that are not cluster-specific ('attribute access' functions moved to this chapter)
1.7	11-June-2013	The following clusters were added for Home Automation: Binary Input (Basic), Door Lock, Illuminance Measurement, Occupancy Sensing. Other minor modifications also made
1.8	14-Aug-2013	Various updates made for ZigBee Light Link release
1.9	14-Oct-2013	Various updates made for Home Automation release, including the addition of EZ-mode Commissioning and modifications to the Identify and Occupancy Sensing clusters

Important Notice

NXP Laboratories UK Ltd
(Formerly Jennic Ltd)
Furnival Street
Sheffield
S1 4QT
United Kingdom

Tel: +44 (0)114 281 2655
Fax: +44 (0)114 281 2951

For the contact details of your local NXP office or distributor, refer to:

www.nxp.com

For online support resources, visit the Wireless Connectivity TechZone:

www.nxp.com/techzones/wireless-connectivity