ZigBee 3.0 Devices User Guide

JN-UG-3131

Revision 2.0

18 November 2019

Contents

Preface

This manual provides information on the ZigBee device types for ZigBee 3.0. These incluse the ZigBee Base Device, required for all ZigBee 3.0 nodes, as well as other device types like the ZigBee Lighting and Occupancy (ZLO), supported by the NXP K32W041, K32W061, and JN518x family of microcontrollers. The clusters used by the device types are detailed elsewhere in the *ZigBee Cluster Library (for ZigBee 3.0) User Guide (JN-UG-3132)*.

Organisation

This manual contains three chapters:

- Chapter 1 introduces ZigBee device types and provides general guidance on implementing device types in ZigBee application software.
- Chapter 2 details the ZigBee Base Device (ZBD), including the associated functions and other resources.
- Chapter 3 details the ZigBee Lighting and Occupancy (ZLO) device types, including the device software structures and functions.

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

ACE	Ancillary Control Equipment
APDU	Application Protocol Data Unit
API	Application Programming Interface
BDB	Base Device Behaviour
CIE	Control and Indicating Equipment
DRLC	Demand-Response and Load Control
HA	Home Automation
IAS	Intruder Alarm System
SDK	Software Developer's Kit
SE	Smart Energy
WD	Warning Device
ZBD	ZigBee Base Device
ZCL	ZigBee Cluster Library
ZLO	ZigBee Lighting and Occupancy
ZPS	ZigBee PRO Stack

Related Documents

JN-UG-3130	ZigBee 3.0 Stack User Guide
JN-UG-3132	ZigBee Cluster Library (for ZigBee 3.0) User Guide
	Connectivity Framework Reference Manual
JN-UG-3135	DK6 Encryption Tool (JET) User Guide
13-0402	Base Device Behavior Specification [from ZigBee Alliance]
15-0014	Lighting & Occupancy Device Specification [from ZigBee Alliance]
075123 rev 6	ZigBee Cluster Library Specification [from ZigBee Alliance]

Support Resources

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

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

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

Trademarks

All trademarks are the property of their respective owners.

Chip Compatibility

The device software described in this manual can be used on the NXP K32W041, K32W061, and JN518x family of wireless microcontrollers.

1. Introduction

The nodes of a ZigBee wireless network are based on device types defined by the ZigBee Alliance, where a device type is a software entity that determines the functionality supported by a node. This chapter introduces ZigBee device types and describes related concepts that are required in programming software applications for ZigBee nodes.

Note: ZigBee device types have previously been collected together in market-specific application profiles, such as Home Automation. ZigBee 3.0 allows devices from different market sectors to exist in the same network. Therefore, application profiles are not so prevalent in ZigBee 3.0 but are still supported for backward compatibility.

1.1 ZigBee Device Types

A device type is a software entity which defines the functionality of a ZigBee node. The device type defines a collection of clusters that make up this functionality. A cluster is therefore a basic building-block of device functionality. Some clusters are mandatory and some are optional. For example, the Thermostat device uses the Basic and Temperature Measurement clusters, and can also use one or more optional clusters.

Note: The clusters used by a device type are supplied in the ZigBee Cluster Library (ZCL). The ZCL is detailed in the *ZigBee Cluster Library (for ZigBee 3.0) User Guide (JN-UG-3132)*.

A device is an instance of a device type.

A network node can support more than one device type. The application for a device type runs on a software entity called an endpoint and each node can have up to 240 endpoints, numbered from 1.

In addition, every ZigBee 3.0 node must employ the following devices:

- **ZigBee Base Device (ZBD):** This is a standard device type which handles fundamental operations such as commissioning. This device does not need an endpoint. The ZigBee Base Device is fully detailed in Chapter 2.
- **ZigBee Device Objects (ZDO):** This represents the ZigBee node type (Co-ordinator, Router or End Device) and has a number of communication roles. This device occupies endpoint 0.

The relative locations of the different devices are indicated in Section 1.2.

1.2 Software Architecture

The basic ZigBee 3.0 software architecture is shown in Figure 1 below, which illustrates the locations of the ZigBee devices.

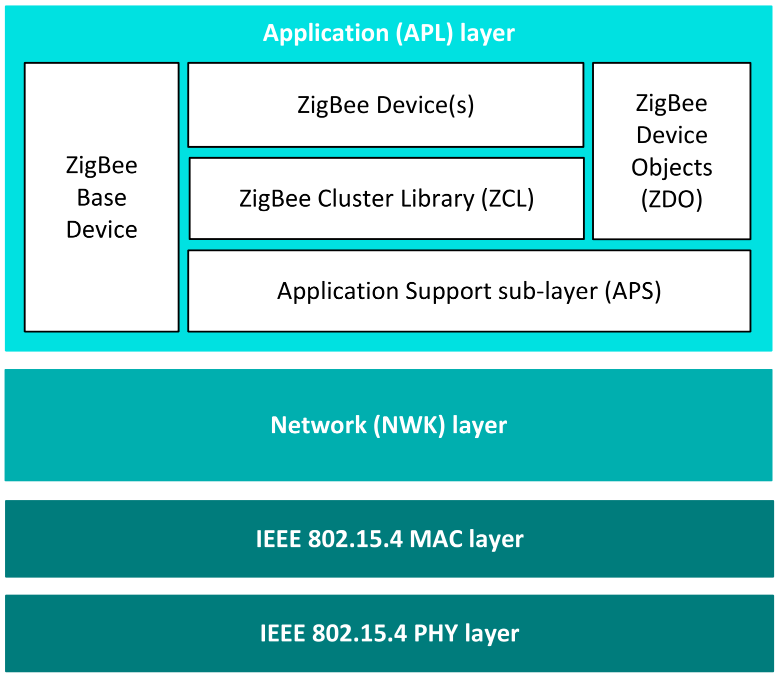


Figure 1: Basic Software Architecture

For more detailed software architecture information, refer to the *ZigBee 3.0 User Guide (JN-UG-3130)*.

1.3 Shared Device Structure

The basic operations in a ZigBee 3.0 network are concerned with reading and setting the attribute values of the clusters of a device. In each device, attribute values are exchanged between the application and the ZigBee Cluster Library (ZCL) by means of a shared structure. This structure is protected by a mutex (described in the *ZCL User Guide (JN-UG-3132)*). The structure for a particular device contains structures for the clusters supported by that device.

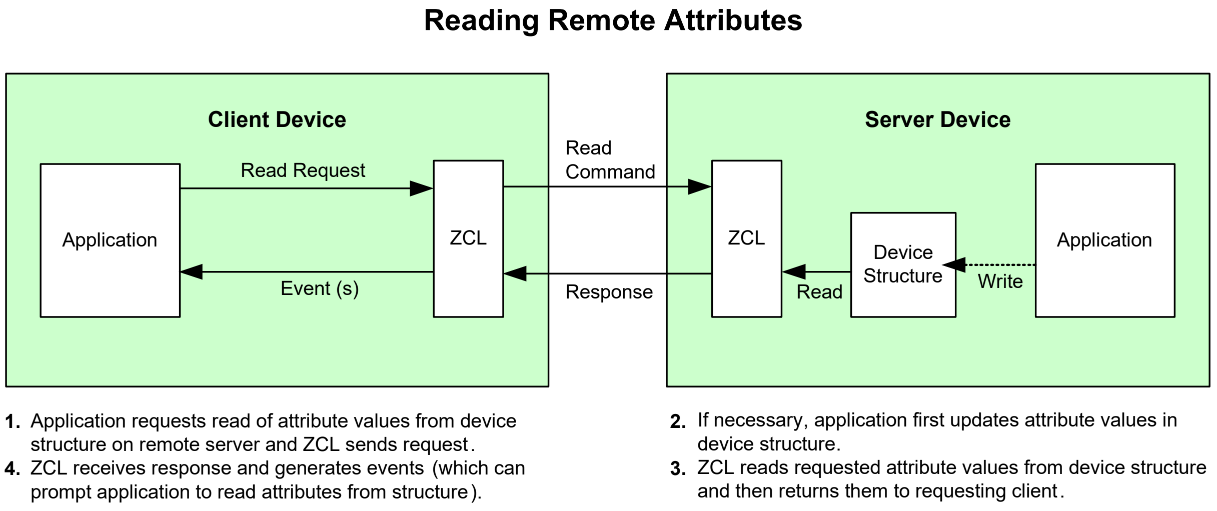
Note: In order to use a cluster which is supported by a device, the relevant option for the cluster must be specified at build-time - see Section 1.6.

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.
- The ZCL parses incoming commands that write attribute values to the structure. The written values can then be read by the local application.

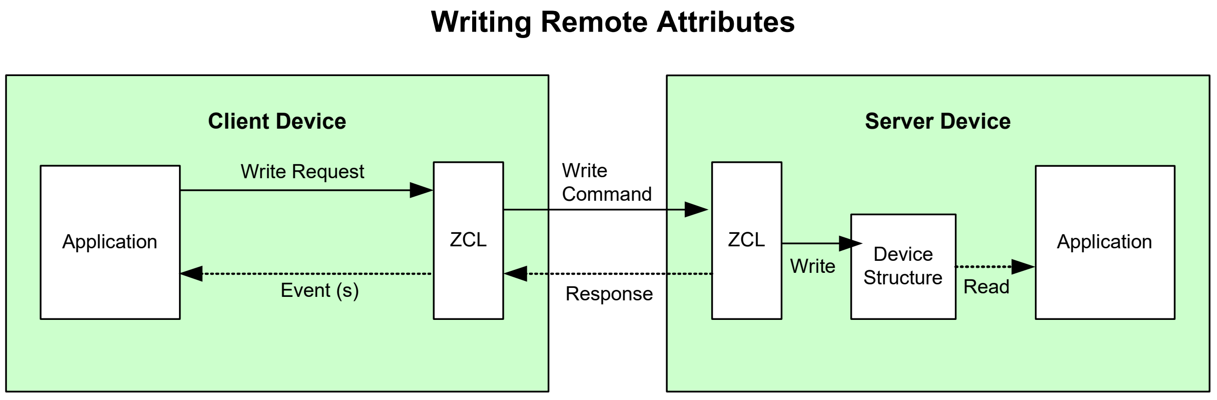
Remote read and write operations involving a shared device structure are illustrated in Figure 2 below. For more detailed descriptions of these operations, refer to the *ZCL User Guide (JN-UG-3132).*

Note: The shared device structure is located on the server device, which hosts the cluster server to be accessed. The client device, which performs the remote access, hosts the corresponding cluster client.



1. Application requests read of attribute values from device structure on remote server and ZCL sends request.
4. ZCL receives response 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. ZCL sends 'write attributes' request to remote server.
5. ZCL can receive optional response and generate events for the application (that indicate any unsuccessful writes).

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

Figure 2: 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).

1.4 Device Initialisation

A ZigBee 3.0 application is initialised as described in the section “Forming and Joining a Network” of the *ZigBee 3.0 Stack User Guide (JN-UG-3130)*. In addition, some device initialisation must be performed.

The initialisation of ZigBee devices must be performed in the following places and order:

1. In the header file **zcl_options.h**, enable the required compile-time options. These options include the clusters to be used by the device, the client/server status of each cluster and the optional attributes for each cluster. For more information on compile-time options, refer to Section 1.6.
2. In the application, create an instance of the device structure by declaring a file scope variable - for example:

   ```
   tsZLO_DimmableLightDevice sDevice;
   ```
3. In the initialisation part of the application, set up the device(s) handled by your code, as follows:
   - **a)** Set the initial values of the cluster attributes to be used by the device - for example:

     ```
     sDevice.sBasicCluster.u8StackVersion = 1;
     sDevice.sBasicCluster....
     ```
   - **b)** After calling **eZCL_Initialise()** and before calling **ZPS_eAplAfInit()**, register the device by calling the relevant device registration function - for example, **eZLO_RegisterDimmableLightEndPoint()**. In this function call, the device must be allocated a unique endpoint (in the range 1-240). In addition, its device structure must be specified as well as a user-defined callback function that will be invoked when an event occurs relating to the endpoint (see Section 1.5). As soon as this function has been called, the shared device structure can be read by another device.
   - **c)** After calling **ZPS_eAplAfInit()**, initialise and start the ZigBee Base Device (ZBD) by calling **BDB_vInit()** and then **BDB_vStart()** - refer to Section 2.1 for more details of ZigBee Base Device initialisation.

Note 1: The set of endpoint registration functions for the different device types are detailed in the device type descriptions - for example, in Chapter 3 for Lighting and Occupancy devices.

Note 2: The device registration functions create instances of all the clusters used by the device, so there is no need to explicitly call the individual cluster creation functions, e.g. **eCLD_IdentifyCreateIdentify()** for the Identify cluster.

1.5 Endpoint Callback Functions

A user-defined callback function must be provided for each endpoint used, where this callback function will be invoked when an event occurs (such as an incoming message) relating to the endpoint. The callback function is registered when the endpoint is registered using the registration function for the device type that the endpoint supports (see Section 1.4) - for example, using the function **eZLO_RegisterOnOffLightEndPoint()** for an On/Off Light device (see Section 3.1).

The endpoint callback function has the type definition given below:

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

where *pCallBackEvent* is a pointer the event.

Note: Events that do not have an associated endpoint are delivered via the general stack-supplied callback function **APP_vGenCallback()**. For example, stack leave and join events can be received by the application through this callback function. Stack events are described in the *ZigBee 3.0 Stack User Guide (JN-UG-3130)*.

1.6 Compile-Time Options

Before a ZigBee 3.0 application can be built, compile-time options must be configured in the header file **zcl_options.h** for the application.

Note 1: Cluster-specific compile-time options are detailed in the cluster descriptions in the *ZCL User Guide (JN-UG-3132)*.

Note 2: In addition, compile-time options for the ZigBee Base Device must be set in the file **bdb_options.h** - see Section 2.10.

Number of Endpoints

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

```
#define BDB_FB_NUMBER_OF_ENDPOINTS   3
```

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

Manufacturer Code

The ZCL allows a manufacturer code to be defined for devices developed by a certain manufacturer. This is a 16-bit value allocated to a manufacturer by the ZigBee Alliance and is set as follows:

```
#define ZCL_MANUFACTURER_CODE   0x1037
```

The above example sets the manufacturer code to the default value of 0x1037 (which belongs to NXP) but manufacturers should set their own allocated value.

Enabled Clusters

All required clusters must be enabled in the options header file. For example, an application for an On/Off Light device that uses all the possible clusters will require the following definitions:

```
#define CLD_BASIC
#define CLD_IDENTIFY
#define CLD_GROUPS
#define CLD_SCENES
#define CLD_ONOFF
```

Server and Client Options

Many clusters have options that indicate whether the cluster will act as a server or a client on the local device. If the cluster has been enabled using one of the above definitions, the server/client status of the cluster must be defined. For example, to employ the Groups cluster as a server, include the following in the header file:

```
#define GROUPS_SERVER
```

Support for Attribute Read/Write

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

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

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

Optional Attributes

Many clusters have optional attributes that may be enabled at compile-time via the options header file - for example, the Basic cluster ‘application version’ attribute is enabled as follows:

```
#define CLD_BAS_ATTR_APPLICATION_VERSION
```

2. ZigBee Base Device

The ZigBee Base Device (ZBD) is a mandatory device on all nodes of a ZigBee 3.0 network. It exists alongside one or more other ZigBee device types on a node, but does not require an endpoint. The ZigBee Base Device provides a framework for the use of ZigBee device types. It implements basic functionality that may be required by all nodes and ensures consistent behaviour across all nodes, particularly regarding network creation and joining as well as network security.

The network commissioning and security functionality of the ZigBee Base Device is described in this chapter, as well as the NXP resources needed to implement these features in ZigBee 3.0 applications on K32W041, K32W061, or JN518x devices. Detailed information about the ZigBee Base Device is provided in the *ZigBee Base Device Behavior Specification (13-0402)*, available from the ZigBee Alliance.

2.1 Initialising and Starting the ZigBee Base Device

The ZigBee Base Device must be initialised in the application code using the function **BDB_vInit()**. This function must be called after initialising the ZigBee PRO stack and after restoring the ZigBee Base Device attribute *bbdbNodeIsOnANetwork* from persistent storage.

Note 1: BDB_vInit() internally calls the function **BDB_vSetKeys()**, which loads into memory the pre-configured link key(s) from the file **bdb_link_keys.c**. Network security and the pre-configured link keys are described in Section 2.3.

Note 2: The ZigBee Base Device requires a number of internal software times, the number defined by the macro BDB_ZTIMER_STORAGE. Therefore, when the application calls **ZTIMER_eInit()** to initialise the required software timers and allocate storage (array elements) for them, it must add BDB_ZTIMER_STORAGE timers for use by the ZigBee Base Device. This function must be called before **BDB_vInit()**. Software timers and their associated functions are described in the *ZigBee 3.0 Stack User Guide (JN-UG-3130)*.

The ZigBee Base Device can then be started by calling the function **BDB_vStart()**. Depending on the node type and whether the node was previously a member of a network, this function may or may not perform an action, as described below. In either case, the function will finally invoke the callback function **APP_vBdbCallback()** with a suitable event.

If the node was not in a network:

For a Router node that supports Touchlink commissioning (see Section 2.2.1), the function selects a radio channel for the node from the set of primary channels for Touchlink specified in the BDBC_TL_PRIMARY_CHANNEL_SET bitmap (see Section 2.5.2.2). Either the first channel of the specified set will be selected or, if the macro RAND_CHANNEL is set to TRUE (in the file **bdb_options.h**), a channel will be selected from the set at random.

For the Co-ordinator and other Router and End Device nodes, no action will be taken and the application will subsequently need to either form a network (Co-ordinator or Router) or join the node to a network using one of the commissioning methods described in Section 2.2 (End Device or Router).

In the above cases, the function will generate a BDB_EVENT_INIT_SUCCESS event.

If the node was in a network:

For Co-ordinator and Router nodes, no action is taken and the function generates a BDB_EVENT_INIT_SUCCESS event.

For an End Device node, the function will attempt to rejoin the node to the network. It will perform a series of rejoin cycles, where each cycle comprises the following three rejoin attempts:

1. First attempt with the previously used network parameters (without network discovery)
2. Second attempt with network discovery on the set of primary channels specified in the *u32bdbPrimaryChannelSet* bitmap (attribute)
3. Third attempt with network discovery on the set of secondary channels specified in the *u32bdbSecondaryChannelSet* bitmap (attribute)

The channel bitmaps are ZigBee Base Device attributes, described in Section 2.5.1.

The above rejoin cycle will be performed up to a maximum of BDBC_IMP_MAX_REJOIN_CYCLES times, which is an implementation-specific ZigBee Base Device constant (see Section 2.5.2).

If a rejoin attempt is successful, the function will generate the event BDB_EVENT_REJOIN_SUCCESS.

If all the rejoin attempts were unsuccessful, the function will generate the event BDB_EVENT_REJOIN_FAILURE unless unsecured joins are enabled through the APS attribute *apsUseInsecureJoin*, in which case the function will attempt a join through Network Steering (described in Section 2.2.2). The nature of the join depends on the value of the Extended PAN ID (EPID) set in the APS attribute *ApsUseExtendedPanid*:

- For a non-zero EPID, the node will attempt to join the network with this EPID
- For a zero EPID, the function will attempt to join any available network

This join will be attempted with an automatic call to the function **BDB_eNsStartNwkSteering()**.

2.2 Network Commissioning

Network commissioning covers the following activities:

- Creating a network
- Allowing devices to join the network (through the local node)
- Joining a network
- Binding a local endpoint to an endpoint on a remote node
- Adding a remote node to a group

The commissioning activities that can be performed by an individual node depend on the ZigBee node type (Co-ordinator, Router, End Device) and the commissioning modes that are enabled for the node. A number of different commissioning modes are available through the ZigBee Base Device. These modes are listed in Table 2 along with the commissioning activities that they support.

Commissioning Mode	**Functionality**
Touchlink	• Creating a new network • Allowing other devices to join an existing network • Joining local device to an existing network
Network Steering	• Allowing other devices to join an existing network • Joining local device to an existing network
Network Formation	• Creating a new network
Finding and Binding	• Binding a local endpoint to an endpoint on a remote node • Adding a remote node to a group

Table 1: Functionality of Commissioning Modes

The commissioning modes are individually enabled/disabled via the attribute *u8bdbCommissioningMode*, as indicated in Table 2 below. This attribute is a bitmap with a bit for each of four commissioning mode - a bit is to ‘1’ to enable or ‘0’ to disable the corresponding commissioning mode. Enumerations are available to enable the individual modes (set their bits to ‘1’).

Bit	**Commissioning Mode**	**Enumeration**
0	Touchlink	BDB_COMMISSIONING_MODE_TOUCHLINK
1	Network Steering	BDB_COMMISSIONING_MODE_NWK_STEERING
2	Network Formation	BDB_COMMISSIONING_MODE_NWK_FORMATION
3	Finding and Binding	BDB_COMMISSIONING_MODE_FINDING_N_BINDING

Table 2: Commissioning Modes (configured via bdbCommissioningMode)

The current commissioning state on a node is reflected in the attribute *ebdbCommissioningStatus*.

In the NXP implementation of the ZigBee Base Device, the individual commissioning modes are initiated under application control using supplied API functions. A commissioning mode can be invoked by the application provided that the mode is enabled and the node type is relevant to the mode (for example, an End Device cannot perform Network Formation).

The commissioning modes are outlined in the sub-sections below. For detailed information on these modes, refer to the *ZigBee Base Device Behavior Specification (13-0402-08)*.

Note: A node will normally be prompted to enter commissioning by a user action, such as pressing a button on the node. This action may be on behalf of the node as a whole or a single endpoint on the node.

2.2.1 Touchlink

Touchlink commissioning can be used to form a new network and/or join a node to an existing network. Touchlink is initiated on a node called the ‘initiator’ which either will be a member of an existing network or (if not) will create a new network. In both cases, the initiator will join a second node to the network, called the ‘target’ node.

Touchlink is provided as a cluster in the ZigBee Cluster Library (ZCL). The initiator must support the Touchlink cluster as a client and the target node must support the cluster as a server. If it is required on a node, Touchlink commissioning must be enabled via the ZigBee Base Device attribute *u8bdbCommissioningMode*. For detailed information on the Touchlink Commissioning cluster and how to implement Touchlink, refer to the *ZigBee Cluster Library User Guide (JN-UG-3132).*

A ‘Touchlink Pre-configured Link Key’ may be provided that is used during the commissioning of a node into a secured network (see Section 2.3).

If Touchlink commissioning is not successful, this is indicated by a status of NO_SCAN_RESPONSE through the attribute *ebdbCommissioningStatus* (all other states indicate success).

2.2.2 Network Steering

Network Steering can be used to join the local node to an existing network or allow other nodes to join a network via the local node.

If Network Steering is required on a node, it must be enabled via the attribute *u8bdbCommissioningMode*. You can start Network Steering from your application by calling the function **BDB_eNsStartNwkSteering()**.

The path taken depends on whether the local node is already a member of a network, as indicated by the Boolean attribute *bbdbNodeIsOnANetwork*. In all cases, the outcome of Network Steering is indicated by events passed into the callback function **APP_vBdbCallback()**.

Node is already in a network

When the node is already a member of a network, it opens the network for other nodes to join for a fixed period of time by broadcasting a Management Permit Joining request (any node type can open the network in this way). This period is 180 seconds by default, but can be configured (in seconds) through the ZigBee Base Device constant BDBC_MIN_COMMISSIONING_TIME (see Section 2.5.2). After initiating the above broadcast, the event BDB_EVENT_NWK_STEERING_SUCCESS will be generated.

Node is not in a network

When the node is not a member of a network and is a Router or End Device, it searches for a suitable network to join and, if it finds one, attempts to join the network, as follows:

1. The node performs a network discovery by scanning the primary set of radio channels specified through the *u32bdbPrimaryChannelSet* bitmap (attribute). If no open network is found, the network discovery is repeated on the secondary set of radio channels specified through the *u32bdbSecondaryChannelSet* bitmap (attribute). If still no network is found, the event BDB_EVENT_NO_NETWORK is generated and the Network Steering is abandoned.
2. If at least one open network was found, the node will then attempt to join each discovered open network one by one, up to a maximum of BDBC_MAX_SAME_NETWORK_RETRY_ATTEMPTS times. If a network is successfully joined, the attribute *bbdbNodeIsOnANetwork* is set to TRUE. If there was no successful join following a scan of the primary channels, the scan is repeated (Step 1) on the secondary channels. If there is still no successful join following this scan, the BDB_EVENT_NWK_JOIN_FAILURE event is generated and the Network Steering is abandoned.
3. The joining node is authenticated and receives the network key from its parent. If the network being joined has centralised security and therefore a Trust Centre, the node unicasts a Node Descriptor request to the Trust Centre. The Node Descriptor received back is checked to ensure that the Trust Centre supports the ZigBee PRO stack version r21 or above. If this is the case, the node performs the procedure for retrieving a new Trust Centre link key to replace its pre-configured link key. Failure at any point will be indicated to the application by a BDB_EVENT_NWK_JOIN_FAILURE event.
4. On successful completion of the above steps, the joining node requests that the ‘permit joining’ time (for new nodes to join the network) is extended by BDBC_MIN_COMMISSIONING_TIME (180s by default) and generates a BDB_EVENT_NWK_STEERING_SUCCESS event for the application.

Depending on the outcome of the above Network Steering process:

- If the node successfully joins a network, you may wish to bind the node to another node or add the node to a group, in which case it is necessary to continue to the Finding and Binding stage, described in Section 2.2.4.
- If the node fails to join a network, you may wish to make sure the desired network is open for joining and re-initiate this Network Steering procedure. In the case of a Router node, the application may opt to form its own distributed network, in which case it is necessary to continue to the Network Formation stage described in Section 2.2.3.

2.2.3 Network Formation

Network Formation allows a new network to be created by a Co-ordinator or Router.

- A Co-ordinator will form a centralised security network (see Section 2.3.1) and activate its Trust Centre functionality.
- A Router will form a distributed security network (see Section 2.3.2).

If Network Formation is required on a node, it must be enabled via the attribute *u8bdbCommissioningMode*. You can start Network Formation from your application by calling the function **BDB_eNfStartNwkFormation()**.

The node will perform a scan of the primary set of radio channels specified through the *u32bdbPrimaryChannelSet* bitmap (attribute) to form a centralised or distributed network with a unique PAN ID on one of the free primary channels. If this network formation fails or the primary channel bitmap is set to zero, the node will perform a scan of the secondary set of radio channels specified through the *u32bdbSecondaryChannelSet* bitmap (attribute) to form a centralised or distributed network with a unique PAN ID on one of the free secondary channels.

During the formation of a distributed security network by a Router:

- The above channel scans will start with the first channel of the relevant set and cover all the specified channels.
- If the macro RAND_CHANNEL is TRUE (in the application), a channel will be selected at random from the scanned channels.
- The macro RAND_DISTRIBUTED_NWK_KEY should be set to TRUE to choose a network key at random (but may be set to FALSE during application development in order to use a specific network key).
- The PAN ID and Extended PAN ID are allocated at random (but must not clash with those of other networks operating in the neighbourhood).
- The 16-bit network address of the local is allocated at random.

In all cases, successful Network Formation is indicated by the event BDB_EVENT_NWK_FORMATION_SUCCESS through the callback function **APP_vBdbCallback()**, while unsuccessful Network Formation is indicated by the event BDB_EVENT_NWK_FORMATION_FAILURE.

If Network Formation is successful, the new network will consist of just one node. Further nodes can be added to the network using Network Steering (see Section 2.2.2) or Touchlink (see Section 2.2.1).

2.2.4 Finding and Binding

Finding and Binding mode allows a node in the network to be paired with another network 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 commissioning mode is to bind an endpoint on a new node to a compatible endpoint on a remote node in the network (depending on the supported clusters). Alternatively, the new node may be added to a group of nodes that are collectively controlled.

If it is required on a node, Finding and Binding must be enabled via the attribute *u8bdbCommissioningMode*.

In Finding and Binding, a node can have one of two roles:

- **Initiator:** This node can either create a (local) binding with a remote endpoint or request that the remote endpoint is added to a group
- **Target:** This node identifies itself, and 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 path followed by the Finding and Binding process depends on whether the local endpoint is an initiator or a target.

2.2.4.1 Initiator Node

Finding and Binding can be started on an initiator node by calling the function **BDB_eFbTriggerAsInitiator()** - this function may be called as the result of a user action on the node, such as a button-press. The initiator will then remain in Finding and Binding mode for a fixed time-interval (in seconds) defined by the constant BDBC_MIN_COMMISSIONING_TIME. If Finding and Binding does not succeed within this time, the event BDB_EVENT_FB_TIMEOUT is generated and passed into the callback function **APP_vBdbCallback()**.

Once Finding and Binding has started, the initiator node searches for target endpoints by broadcasting an Identify Query command periodically with a period (in seconds) defined through the macro BDB_FB_RESEND_IDENTIFY_QUERY_TIME.

Note: Before each broadcast attempt, the event BDB_EVENT_FB_NO_QUERY_RESPONSE is generated and passed into **APP_vBdbCallback()**, in order to give the application the opportunity to exit the current Finding and Binding process (see below).

If the initiator receives an Identify Query response from a remote endpoint, the application must pass the ZCL event BDB_E_ZCL_EVENT_IDENTIFY_QUERY to the Base Device using the function **BDB_vZclEventHandler()**. This will allow the Base Device to gather information about the identifying device by sending a Simple Descriptor request to the relevant endpoint. If the requested Simple Descriptor is then successfully received back, the callback function checks this descriptor for clusters that match those on the initiator. The application is notified of this via a

BDB_EVENT_FB_HANDLE_SIMPLE_DESC_RESP_OF_TARGET event passed into **APP_vBdbCallback()**.

If there is at least one matching cluster, the initiator does one of the following:

- If binding is required (indicated by the *u16bdbCommissioningGroupID* attribute being equal to 0xFFFF), the initiator adds the remote endpoint to the local Binding table (but may first need to request the IEEE/MAC address of the remote node).
- If grouping is required (indicated by the *u16bdbCommissioningGroupID* attribute being equal to a 16-bit group address), the initiator will request that the target endpoint adds the group address to its Group Address table.

The application is notified of a successful binding or grouping via the following events:

- For a binding:
  - BDB_EVENT_FB_BIND_CREATED_FOR_TARGET for success
  - BDB_EVENT_FB_ERR_BINDING_FAILED for failure
- For a grouping:
  - BDB_EVENT_FB_GROUP_ADDED_TO_TARGET for success
  - BDB_EVENT_FB_ERR_GROUPING_FAILED for failure

At this point, the application can remotely stop identification mode (and therefore Finding and Binding) on the target node by calling the Identify cluster function **eCLD_IdentifyCommandIdentifyRequestSend()** to request that the identification mode period is set to zero.

A Finding and Binding process can be stopped on the initiator endpoint using the function **BDB_vFbExitAsInitiator()**. This function is typically called in the callback function **APP_vBdbCallback()** as the result of a user action, such as a button-press or button-release.

2.2.4.2 Target Node

Finding and Binding can be started on a target node by calling the function **BDB_eFbTriggerAsTarget() -** this function may be called as the result of a user action on the node, such as a button-press.

The target node then uses the Identify cluster to put itself into identification mode for a fixed period of time. This period (in seconds) is determined by `u16IdentifyTime`, an Identify cluster attribute which is automatically set to the value of the constant BDBC_MIN_COMMISSIONING_TIME. In identification mode, the cluster will respond to any received Identify Query commands, as well as other Finding and Binding commands. The node may also visually or audibly indicate that it is in identification mode. On exiting identification mode at the end of the above period, the cluster will no longer be able to process Identify Query commands but the node will still be able to service other commands from the initiator related to the binding/grouping. The Identify cluster is fully described in the *ZigBee Cluster Library User Guide (JN-UG-3132)*.

A target node can be brought out of the Finding and Binding process in either of the following ways:

- The local application can call the function **BDB_vFbExitAsTarget()** as the result of a user action, such as a button-press or button-release.
- The remote application (on the initiator) can call the Identify cluster function **eCLD_IdentifyCommandIdentifyRequestSend()** to request that the identification mode period is set to zero. To indicate to the Base Device that the identification process has ended, the application must pass the ZCL event BDB_E_ZCL_EVENT_IDENTIFY to the Base Device using the **BDB_vZclEventHandler()** function. This will allow the Base Device to exit the 'Finding and Binding' process on the target endpoint.

2.2.5 Out-Of-Band Commissioning

A node can be commissioned to a ZigBee network via out-of-band means - that is, not using IEEE802.15.4 packets operating in the radio channel used by the target network. For example, the out-of-band commissioning could be conducted from another ZigBee device using inter-PAN packets (operating in a different radio channel) or by a commissioning device that uses NFC (Near Field Communication).

Out-of-band commissioning can be used to create a new network by starting the Co-ordinator or to join a Router or End Device to an existing network. To do this, commissioning data must be sent to the node via an out-of-band means. This data includes details of the network (see Section 2.7.5). The application must pass the received commissioning data to the ZigBee Base Device and start out-of-band commissioning using the function **BDB_u8OutOfBandCommissionStartDevice()**. The data is then stored locally.

As part of the out-of-band commissioning of a node to an existing centralised network, the Trust Centre of the joined network may need to validate the new node by checking that the node contains appropriate data values, such as the correct network key and Trust Centre address. If such a validation request is received by the node, the required data values can be obtained by the application in either of two ways:

- The function **BDB_vOutOfBandCommissionGetData()** can be used to read the relevant data values. In this case, the application should encrypt the obtained network key before sending the data to the Trust Centre. The install code for the node should be used in this encryption.
- The function **BDB_eOutOfBandCommissionGetDataEncrypted()** can be used to read the relevant data values and encrypt the obtained network key - thus, the network key will be delivered already encrypted. The install code for the node to be used in this encryption must be specified in the function call. The application can then send the obtained data to the Trust Centre.

Once the Trust Centre has received the requested data, it can decrypt the obtained network key using the function **BDB_bOutOfBandCommissionGetKey()** and then check that the correct key is being used. This function requires the install code for the new node, which must be supplied to the Trust Centre via out-of-band means (for example, via a keypad).

Security keys and install codes are described in Section 2.3.

2.3 Network Security

The ZigBee Base Device supports the following network security modes:

- Centralised security
- Distributed security

These are described in the sub-sections below.

All Router and End Device nodes should support both centralised security and distributed security by adapting to the security scheme employed by the network that they join. A Co-ordinator supports only centralised security.

When the application calls **BDB_vInit()**, this function internally calls the function **BDB_vSetKeys()**. This function will load the appropriate pre-configured link key(s), depending on whether the node type supports centralised and/or distributed security. The pre-configured link keys are defined in the file **bdb_link_keys.c**.

2.3.1 Centralised Security Networks

A centralised security network is formed by a Co-ordinator, which also acts as the Trust Centre for the network. When a node attempts to join the network, it is authenticated by this Trust Centre before it is allowed into the network.

For participation in centralised security networks, all nodes must be pre-configured with a link key. This key is used to encrypt the network key when passing it from the Trust Centre to a newly joined node. When a node joins a network with centralised security, the ZigBee Base Device will automatically use the relevant pre-configured link key. This will also be the case for a Co-ordinator that forms a new centralised security network.

The following key types can be pre-configured for centralised security:

- **Default Global Trust Centre Link Key:** This key is factory-programmed into all nodes and is used to encrypt communications between the Trust Centre and a joining node.
- **Touchlink Pre-configured Link Key:** This key is factory-programmed into all nodes that can employ Touchlink commissioning and is used to encrypt communications between the Router parent and a joining node. The Touchlink Pre-configured Link Key can be one of three types:
    - Development key, used during development before ZigBee certification
    - Master key, used after successful ZigBee certification
    - Certification key, used during ZigBee certification testing

  The link key used in the final products should be a 'master key', which results from the successful ZigBee certification of the product.
- **Install Code-derived Pre-configured Link Key:** This key is derived by the ZigBee stack from a random install code which is assigned to each Router and End Device node in the factory. The install code is factory-programmed into the node but provided to the Trust Centre via out-of-band means when the node is commissioned. The use of install codes is described in more detail below.

Install Codes

An install code can be used to create an initial link key employed in commissioning an individual node into a centralised security network. An install code is assigned to the node in the factory. It is a random code but is not necessarily unique (the same install code may be randomly generated for more than one node). The ZigBee stack derives a link key from the install code using a Matyas-Meyer-Oseas hash function. The install code is factory-programmed into the node and also accompanies the node (e.g. in printed form) when it leaves the factory. The process of using an install code to commission a node is outlined below.

In the factory:

1. An install code is randomly generated for the individual node.
2. The install code is programmed into the node.
3. A pre-configured link key is derived from the install code by the ZigBee stack.
4. The install code is shipped with the node (by some unspecified means).

During installation:

5. The install code that was shipped with the node is installed into the Co-ordinator/Trust Centre.
6. The pre-configured link key is derived from the install code by the ZigBee stack of the Co-ordinator/Trust Centre.
7. The Trust Centre and node subsequently use the pre-configured link key in joining the node to the network (e.g. to encrypt/decrypt the network key).

More detailed information about install codes can be found in the *ZigBee Base Device Behavior Specification (13-0402-08)*.

2.3.2 Distributed Security Networks

A distributed security network is formed by a Router and does not have a Trust Centre. It consists only of Routers and End Devices. When a node attempts to join the network, it is authenticated by its Router parent before it is allowed into the network.

For participation in distributed security networks, all Router and End Device nodes must be pre-configured with a link key. This key is used to encrypt the network key when passing it from a Router parent to a newly joined node. When a Router or End Device joins a network with distributed security, the ZigBee Base Device will automatically use the relevant pre-configured link key. This will also be the case for a Router that forms a new distributed security network.

The following key types can be pre-configured for distributed security:

- **Distributed Security Global Link Key:** This key is factory-programmed into all nodes and is used to encrypt communications between the Router parent and a joining node.
- **Touchlink Pre-configured Link Key:** This key is factory-programmed into all nodes that can employ Touchlink commissioning and is used to encrypt communications between the Router parent and a joining node. The Touchlink Pre-configured Link Key can be one of three types:
  - Development key, used during development before ZigBee certification
  - Master key, used after successful ZigBee certification
  - Certification key, used during ZigBee certification testing

  The link key used in the final products should be a ‘master key’, which results from the successful ZigBee certification of the product.

2.4 ZigBee Base Device Rejoin Handling

For a Router or End Device, there are instances in which the ZigBee PRO stack will initiate a network rejoin attempt. These include:

- A Router or End Device which receives a ‘leave with rejoin’ request
- An End Device which polls its parent for data but fails to receive a response

The ZigBee Base Device handles the stack events that result from this rejoin attempt:

- If the stack event ZPS_EVENT_NWK_FAILED_TO_JOIN is received to indicate an unsuccessful rejoin, the ZigBee Base Device makes a series of rejoin attempts as described for the case “If the node was in a network” in Section 2.1. If a rejoin attempt is successful, the event BDB_EVENT_REJOIN_SUCCESS is generated to notify the application. If all rejoins are unsuccessful, the event BDB_EVENT_REJOIN_FAILURE is generated unless unsecured joins are enabled, in which case a join through Network Steering is attempted.
- If the stack event ZPS_EVENT_NWK_JOINED_AS_ROUTER or ZPS_EVENT_NWK_JOINED_AS_END_DEVICE is received to indicate a successful rejoin, the event BDB_EVENT_REJOIN_SUCCESS is generated to notify the application.

2.5 Attributes and Constants

2.5.1 Attributes

The attributes of the ZigBee Base Device are contained in the structure `BDB_tsAttrib`, shown below.

```
typedef struct
{
    uint16                          u16bdbCommissioningGroupID;
    uint8                           u8bdbCommissioningMode;
    BDB_teCommissioningStatus       ebdbCommissioningStatus;
    uint64                          u64bdbJoiningNodeEui64;
    uint8                           au8bdbJoiningNodeNewTCLinkKey[16];
    bool_t                          bbdbJoinUsesInstallCodeKey;
    const uint8                     u8bdbNodeCommissioningCapability;
    bool_t                          bbdbNodeIsOnANetwork;
    uint8                           u8bdbNodeJoinLinkKeyType;
    uint32                          u32bdbPrimaryChannelSet;
    uint8                           u8bdbScanDuration;
    uint32                          u32bdbSecondaryChannelSet;
    uint8                           u8bdbTCLinkKeyExchangeAttempts;
    uint8                           u8bdbTCLinkKeyExchangeAttemptsMax;
    uint8                           u8bdbTCLinkKeyExchangeMethod;
    uint8                           u8bdbTrustCenterNodeJoinTimeout;
    bool_t                          bbdbTrustCenterRequireKeyExchange;
    bool_t                          bTLStealNotAllowed;
    bool_t                          bLeaveRequested;
}BDB_tsAttrib;
```

The ZigBee Base Device attribute values can be initialised at compile-time in the **bdb_options.h** file using the macros listed in Table 3 below (for information on compile-time options, refer to Section 2.10). The attributes can written to or read at run-time through the above structure.

Note: Both `bTLStealNotAllowed` and `bLeaveRequested` are NXP proprietary variables and not ZigBee attributes.

Attribute	Initialisation Macro
`u16bdbCommissioningGroupID`	BDB_COMMISSIONING_GROUP_ID
`u8bdbCommissioningMode`	BDB_COMMISSIONING_MODE
`ebdbCommissioningStatus`	BDB_COMMISSIONING_STATUS
`u64bdbJoiningNodeEui64`	BDB_JOINING_NODE_EUI64
`au8bdbJoiningNodeNewTCLinkKey[16]`	-
`bbdbJoinUsesInstallCodeKey`	BDB_JOIN_USES_INSTALL_CODE_KEY
`u8bdbNodeCommissioningCapability`	-
`bbdbNodeIsOnANetwork`	-
`u8bdbNodeJoinLinkKeyType`	BDB_NODE_JOIN_LINK_KEY_TYPE
`u32bdbPrimaryChannelSet`	BDB_PRIMARY_CHANNEL_SET
`u8bdbScanDuration`	BDB_SCAN_DURATION
`u32bdbSecondaryChannelSet`	BDB_SECONDARY_CHANNEL_SET
`u8bdbTCLinkKeyExchangeAttempts`	BDB_TC_LINK_KEY_EXCHANGE_ATTEMPTS
`u8bdbTCLinkKeyExchangeAttemptsMax`	BDB_TC_LINK_KEY_EXCHANGE_ATTEMPTS_MAX
`u8bdbTCLinkKeyExchangeMethod`	BDB_TC_LINK_KEY_EXCHANGE_METHOD
`u8bdbTrustCenterNodeJoinTimeout`	BDB_TRUST_CENTER_NODE_JOIN_TIMEOUT
`bbdbTrustCenterRequireKeyExchange`	BDB_TRUST_CENTER_REQUIRE_KEYEXCHANGE
`bTLStealNotAllowed`	-
`bLeaveRequested`	-

Table 3: ZBD Attributes and Initialisation Macros

The attributes are individually described below. For further details, refer to the *ZigBee Base Device Behavior Specification (13-0402-08)*.

u16bdbCommissioningGroupID

This attribute can only be used on a Finding and Binding initiator endpoint. It contains the identifier of the group in which the initiator will put the target endpoints. If it is equal to 0xFFFF, individual (rather than group) bindings will be created. The value of this attribute can be initialised at compile-time using the macro BDB_COMMISSIONING_GROUP_ID.

Use of this attribute requires Finding and Binding to be enabled in the `u8bdbCommissioningMode` attribute.

The Finding and Binding commissioning mode is described in Section 2.2.4.

u8bdbCommissioningMode

This attribute is a bitmap used to indicate which commissioning modes are enabled on an endpoint, where each bit corresponds to a commissioning mode and is set (to '1') when the mode is enabled - this means that the node will be able to implement this commissioning mode, if required. The value of this attribute can be initialised at compile-time using the macro BDB_COMMISSIONING_MODE. The bitmap is illustrated in the table below, along with the enumerations used to set the bits.

Bit	Commissioning Mode	Enumeration
0	Touchlink	BDB_COMMISSIONING_MODE_TOUCHLINK
1	Network Steering	BDB_COMMISSIONING_MODE_NWK_STEERING
2	Network Formation	BDB_COMMISSIONING_MODE_NWK_FORMATION
3	Finding and Binding	BDB_COMMISSIONING_MODE_FINDING_N_BINDING
4-7	Reserved (set to '0')	-

Table 4: bdbCommissioningMode Bitmap

The commissioning modes are described in Section 2.2.

Note: The attribute is used on all node types. However, in order to enable a commissioning mode, it must be available on the node, as indicated through the attribute `u8bdbNodeCommissioningCapability`. The enabled commissioning modes will be a subset of the commissioning capabilities of the node.

ebdbCommissioningStatus

This attribute indicates the status of the commissioning process that is currently underway on an endpoint. The attribute takes one of the values defined in the `BDB_teCommissioningStatus` enumerations (see Section 2.8.2). The attribute is used on all node types. The value of this attribute is updated internally by the ZigBee Base Device implementation, but can be read by the application.

u64bdbJoiningNodeEui64

This attribute contains the 64-bit IEEE/MAC address of a node that is in the process of joining a centralised security network. It is used on the network Co-ordinator only. The value of this attribute is updated internally by the ZigBee Base Device implementation.

au8bdbJoiningNodeNewTCLinkKey

This attribute contains a new link key for use with a node that is currently joining the network but has not yet been granted full network membership. The value of this attribute is updated internally by the ZigBee Base Device implementation (on a joining node and its parent).

bbdbJoinUsesInstallCodeKey

This attribute indicates whether a pre-configured link key must be available for a node before it is allowed to join the network - this may be a pre-installed link key or may be derived from an install code. A value of TRUE means that a link key is required, while FALSE means that a link key is not required. It is used on the network Co-ordinator/ Trust Centre only. The value of this attribute can be initialised at compile-time using the macro BDB_JOIN_USES_INSTALL_CODE_KEY. By default, the attribute should be set to FALSE. The attribute is not used by the ZigBee Base Device and if the attribute is set to TRUE, it is the responsibility of the application to handle this functionality directly and to set the required key (see u8bdbNodeJoinLinkKeyType).

u8bdbNodeCommissioningCapability

This attribute is a bitmap indicating the commissioning capabilities of the node, where each bit corresponds to a commissioning capability and is set (to ‘1’) if the capability is present. The attribute is used on all node types. The application cannot write directly to these bits - they are set according to the options defined in the application makefile. The bitmap and the related makefile options are detailed in the table below.

Bit	Capability	Makefile Options
0	Network Steering	Is set to ‘1’ if BDB_SUPPORT_NWK_STEERING is defined
1	Network Formation	Is set to ‘1’ if BDB_SUPPORT_NWK_FORMATION is defined
2	Finding and Binding	Is set to ‘1’ if either of the following is defined: • BDB_SUPPORT_FIND_AND_BIND_INITIATOR • BDB_SUPPORT_FIND_AND_BIND_TARGET
3	Touchlink	Is set to ‘1’ if any of the following is defined: • BDB_SUPPORT_TOUCHLINK_INITIATOR_END_DEVICE • BDB_SUPPORT_TOUCHLINK_INITIATOR_ROUTER • BDB_SUPPORT_TOUCHLINK_TARGET
4-7	Reserved (set to ‘0’)	-

Table 5: bdbCommissioningCapability Bitmap

The above commissioning modes are described in Section 2.2.

Note: In order to use one of the available commissioning modes, the mode must also be enabled through the attribute `u8bdbCommissioningMode`. The enabled commissioning modes will be a subset of the commissioning capabilities of the node.

bbdbNodeIsOnANetwork

This attribute indicates whether the local node is currently a member of a network. A value of TRUE means that it is in a network (but not necessarily bound to any remote nodes), while FALSE means that it is not in a network. The attribute is used on all node types but the ZigBee Base Device does not maintain it. The application is responsible for persisting the attribute value and initialising the attribute following a power-cycle (before any other ZigBee Base Device functions are called).

u8bdbNodeJoinLinkKeyType

This attribute indicates the type of link key with which the node is able to decrypt the encypted network key received over-air when the node joins a new network. The attribute is used by Router and End Device nodes. The attribute values and the corresponding link key types are listed in the table below, as well as the macros that can be used to define the link keys.

Value	Link Key Type	Link Key Definition Macro
0x00	Default Global Trust Centre Link Key	DEFAULT_GLOBAL_TRUST_CENTER_LINK_KEY
0x01	Distributed Security Global Link Key	DISTRIBUTED_SECURITY_GLOBAL_LINK_KEY
0x02	Install Code Derived Pre-configured Link Key	INSTALL_CODE_DERIVED_PRECONFIGURED_LINK_KEY
0x03	Touchlink Pre-configured Link Key	TOUCHLINK_PRECONFIGURED_LINK_KEY

Table 6: bdbNodeJoinLinkKeyType Values and Macros

u32bdbPrimaryChannelSet

This attribute specifies the primary (first-choice) set of 2.4GHz radio channels that will be used in channel scans. The attribute is a bitmap in which each bit corresponds to a channel and should be set to ‘1’ if the channel is to be included in a scan. The bit number corresponds directly to the channel number - for example, bit 11 corresponds to the 2.4GHz channel 11 and bit 26 corresponds to channel 26. This attribute is used on all node types. The value of this attribute can be initialised at compile-time using the macro BDB_PRIMARY_CHANNEL_SET.

u8bdbScanDuration

This attribute determines the duration of a scan operation per 2.4GHz radio channel. The actual scan duration is calculated from the attribute value as follows:

$$\text{aBaseSuperframeDuration} \times (2^{\text{bdbScanDuration}} + 1)$$

where aBaseSuperframeDuration is defined in the IEEE 802.15.4 specification

The attribute is used on all node types. The value of this attribute is taken from the Scan Duration Time set in the ZPS Configuration Editor.

u32bdbSecondaryChannelSet

This attribute specifies the secondary (second-choice) set of 2.4GHz radio channels that will be used in channel scans. This channel set will be used if the scan of primary channels is unsuccessful. The attribute is a bitmap in which each bit corresponds to a channel and should be set to ‘1’ if the channel is to be included in a scan. The bit number corresponds directly to the channel number - for example, bit 11 corresponds to the 2.4GHz channel 11 and bit 26 corresponds to channel 26. If a scan of secondary channels is not required, the attribute should be set to zero. The attribute is used on all node types. The value of this attribute can be initialised at compile-time using the macro BDB_SECONDARY_CHANNEL_SET.

u8bdbTCLinkKeyExchangeAttempts

This attribute indicates the number of attempts to request a new link key that were made when the node joined the network. The attribute is used on Router and End Device nodes. The value of this attribute can be initialised at compile-time using the macro BDB_TC_LINK_KEY_EXCHANGE_ATTEMPTS.

u8bdbTCLinkKeyExchangeAttemptsMax

This attribute specifies the maximum number of key establishment attempts that will be made before key establishment is abandoned when the node joins a new network. The attribute is used on Router and End Device nodes. The value of this attribute can initialised at compile-time using the macro BDB_TC_LINK_KEY_EXCHANGE_ATTEMPTS_MAX.

u8bdbTCLinkKeyExchangeMethod

This attribute specifies the method that was used to obtain a new link key when the node joined the network. The attribute values and corresponding methods are listed in the table below. This attribute is used on Router and End Device nodes.

Value	Key Exchange Method
0x00	APS Request Key
0x01	Certificate Based Key Exchange (CBKE)
0x02-0xFF	Reserved

Table 7: bdbTCLinkKeyExchangeMethod Values

The value of this attribute can be initialised at compile-time using the macro BDB_TC_LINK_KEY_EXCHANGE_METHOD. It should be initialised to 0x00 (APS Request Key).

u8bdbTrustCenterNodeJoinTimeout

This attribute specifies a timeout (in seconds) for the Trust Centre to delete the Trust Centre-generated link key for a newly joined node when key establishment with the node was unsuccessful. The attribute is used on the network Co-ordinator/Trust Centre only. The value of this attribute can be initialised at compile-time using the macro BDB_TRUST_CENTER_NODE_JOIN_TIMEOUT.

bbdbTrustCenterRequireKeyExchange

This attribute specifies whether the Trust Centre requires a joining node to replace its initial link key with a new link key generated by the Trust Centre. A value of TRUE means that the joining node must successfully complete the link key exchange procedure and failure to do so will result in the node being removed from the network. A value of FALSE means that the joining node will be allowed remain in the network even if it does not successfully complete the link key exchange procedure. The attribute is used on the network Co-ordinator/Trust Centre only. The value of this attribute can be initialised at compile-time using the macro BDB_TRUST_CENTER_REQUIRE_KEYEXCHANGE. It should be initialised according to the Trust Centre policy that is implemented in the network - by default, set it to FALSE for backward compatibility.

bTLStealNotAllowed

This is an NXP proprietary flag which the application can set to prevent Touchlink commissioning commands from another node in a different network from 'stealing' the local node. Clearing the flag allows the node to be stolen, in which case it leaves the current network and either joins the other network or forms a new distributed network, as instructed by Touchlink initiator.

bLeaveRequested

This is an NXP proprietary flag which the application should only read and not write to. If Touchlink commissioning operations cause the ZigBee Base Device to initiate a network leave then this flag is set by the Base Device. When a ZPS_EVENT_NWK_LEAVE_CONFIRM stack event is generated, the application should read this flag and if it reads as TRUE, the application should not handle the event (since the ZigBee Base Device will handle it).

2.5.2 Constants

The ZigBee Base Device constants are divided into two categories:

- Constants used on all nodes - see Section 2.5.2.1
- Constants used on nodes that support Touchlink - see Section 2.5.2.2

2.5.2.1 General Constants

The table below lists the ZigBee Base Device constants that can be used on all nodes and also shows the corresponding macros used to define the constant values in the **bdb_options.h** file.

Constant	Macro
bdbcMaxSameNetworkRetryAttempts	BDBC_MAX_SAME_NETWORK_RETRY_ATTEMPTS
bdbcMinCommissioningTime	BDBC_MIN_COMMISSIONING_TIME
bdbcRecSameNetworkRetryAttempts	BDBC_REC_SAME_NETWORK_RETRY_ATTEMPTS
bdbcTCLinkKeyExchangeTimeout	BDBC_TC_LINK_KEY_EXCHANGE_TIMEOUT

Table 8: ZBD General Constants and Macros

bdbcMaxSameNetworkRetryAttempts

This constant specifies the maximum number of join or key exchange attempts that the node can make on the same network. The value of this constant is defined using the macro BDBC_MAX_SAME_NETWORK_RETRY_ATTEMPTS and should be set to 10 (as recommended in the ZigBee BDB Specification).

bdbcMinCommissioningTime

This constant specifies the minimum time-interval (in seconds) for which a network will be open to allow new nodes to join or for a device to identify itself. The value of this constant is defined using the macro BDBC_MIN_COMMISSIONING_TIME and should be set to 180 (as recommended in the ZigBee BDB Specification).

bdbcRecSameNetworkRetryAttempts

This constant specifies the recommended number of join or key exchange attempts that the node can make on the same network. The value of this constant is defined using the macro BDBC_REC_SAME_NETWORK_RETRY_ATTEMPTS and should be set to 3 (as recommended in the ZigBee BDB Specification).

bdbcTCLinkKeyExchangeTimeout

This constant specifies the maximum time (in seconds) for which a joining node will wait for a response after an APS key request has been sent to the Trust Centre. The value of this constant is defined using the macro BDBC_TC_LINK_KEY_EXCHANGE_TIMEOUT and should be set to 5 (as recommended in the ZigBee BDB Specification).

2.5.2.2 Touchlink Constants

The table below lists the ZigBee Base Device constants that can be used on nodes that support Touchlink commissioning and also shows the corresponding macros used to define the constant values in the **bdb_options.h** file.

Constant	Macro
bdbcTLInterPANTransIdLifetime	BDBC_TL_INTERPAN_TRANS_ID_LIFETIME
bdbcTLMinStartupDelayTime	BDBC_TL_MIN_STARTUP_DELAY_TIME
bdbcTLPrimaryChannelSet	BDBC_TL_PRIMARY_CHANNEL_SET
bdbcTLRxWindowDuration	BDBC_TL_RX_WINDOW_DURATION
bdbcTLScanTimeBaseDuration	BDBC_TL_SCAN_TIME_BASE_DURATION_MS
bdbcTLSecondaryChannelSet	BDBC_TL_SECONDARY_CHANNEL_SET

Table 9: ZBD Touchlink Constants and Macros

bdbcTLInterPANTransIdLifetime

This constant specifies the maximum length of time (in seconds) that an inter-PAN transaction ID remains valid. The value of this constant is defined using the macro BDBC_TL_INTERPAN_TRANS_ID_LIFETIME and should be set to 8 (as recommended in the ZigBee BDB Specification).

bdbcTLMinStartupDelayTime

This constant specifies the length of time (in seconds) that a Touchlink initiator waits for the target to complete its network start-up procedure. The value of this constant is defined using the macro BDBC_TL_MIN_STARTUP_DELAY_TIME and should be set to 2 (as recommended in the ZigBee BDB Specification).

bdbcTLPrimaryChannelSet

This constant specifies the bitmap for the primary (first-choice) set of 2.4GHz radio channels that will be used for a non-extended Touchlink scan. The value of this constant is defined using the macro BDBC_TL_PRIMARY_CHANNEL_SET and should be set to 0x02108800, corresponding to channels 11, 15, 20 and 25 (as recommended in the ZigBee BDB Specification).

bdbcTLRxWindowDuration

This constant specifies the maximum duration (in seconds) that the node's radio receiver remains enabled during Touchlink commissioning, in order to receive responses. The value of this constant is defined using the macro BDBC_TL_RX_WINDOW_DURATION and should be set to 5 (as recommended in the ZigBee BDB Specification).

bdbcTLScanTimeBaseDuration

This constant specifies the base duration (in milliseconds) for which the node's radio receiver will remain enabled after transmitting a scan request during a Touchlink scan operation, in order to receive responses. The value of this constant is defined using the macro BDBC_TL_SCAN_TIME_BASE_DURATION_MS and should be set to 250 (as recommended in the ZigBee BDB Specification).

bdbcTLSecondaryChannelSet

This constant specifies the bitmap for the secondary (second-choice) set of 2.4GHz radio channels that will be used for an extended Touchlink scan. It should contain the channels that remain from those specified in *bdbcTLPrimaryChannelSet*. The value of this constant is defined using the macro BDBC_TL_SECONDARY_CHANNEL_SET and should be set to 0x07FFF800 XOR BDBC_TL_PRIMARY_CHANNEL_SET.

2.6 Functions

This section details the C functions that are provided for the ZigBee Base Device. The functions are listed below along with page references to their descriptions.

Function	Page
BDB_vInit	42
BDB_vSetKeys	43
BDB_vStart	44
BDB_eNfStartNwkFormation	45
BDB_eNsStartNwkSteering	46
BDB_eFbTriggerAsInitiator	47
BDB_vFbExitAsInitiator	48
BDB_eFbTriggerAsTarget	49
BDB_vFbExitAsTarget	50
BDB_bIsBaseIdle	51
BDB_u8OutOfBandCommissionStartDevice	52
BDB_vOutOfBandCommissionGetData	53
BDB_eOutOfBandCommissionGetDataEncrypted	54
BDB_bOutOfBandCommissionGetKey	55

Note 1: The application must provide a user-defined callback function, **APP_vBdbCallback()**, to handle ZigBee Base Device events. The prototype for this function is given in Section 2.9.

Note 2: The ZigBee Base Device supplies the callback function **BDB_vZclEventHandler()** which handles certain ZCL events during the Finding and Binding process, as indicated in Section 2.2.4.

BDB_vInit

void BDB_vInit(BDB_tsInitArgs ***psInitArgs*****);**

Description

This function initialises the ZigBee Base Device (ZBD) and must be the first ZigBee Base Device function called in your code. The function must be called after initialising the ZigBee PRO stack via a call to **ZPS_eAplAfInit()**. The ZigBee Base Device attribute *bbdbNodeIsOnANetwork* must also be restored from persistent storage (if relevant) before calling this function.

Note: Before calling this function, the application must initialise the required ZigBee software timers using the function **ZTIMER_eInit()** from the ZigBee PRO Stack libraries. In doing so, it must add a number of timers for internal use by the ZigBee Base Device, where this number is defined by the macro BDB_ZTIMER_STORAGE.

The initialisation performed by this function includes the following:

- Sets the ZigBee Base Device attributes to their default values, unless other values are defined by the application in the file **bdb_options.h**
- Registers the ZigBee Base Device message queue passed into this function - this message queue is used by the ZigBee Base Device to capture stack events
- Calls **BDB_vSetKeys()** to set the initial pre-configured security keys (defined in the file **bdb_link_keys.c**), according to the node type:
    - For a Co-ordinator, the Default Global Trust Centre Link Key is set
    - For a Router or End Device, both the Default Global Trust Centre Link Key and Distributed Security Global Link Key are set
- Opens timers for ZigBee Base Device internal use

For more information on the security keys, refer to Section 2.3.

Parameters

psInitArgs	Handle of the ZigBee Base Device event queue

Returns

None

BDB_vSetKeys

```
void BDB_vSetKeys(void);
```

Description

This function loads into memory the appropriate pre-configured link key(s) on the local node for the initial security state of the node. The function is automatically called by **BDB_vInit()**. However, it may need to be called explicitly to restore the link keys after a reset which removes the keys from memory.

The type of link key that is loaded depends on the node type, as follows:

- On a Co-ordinator, the Default Global Trust Centre Link Key is loaded for participation in a centralised security network
- On a Router or End Device, both of the following keys are loaded:
  - Default Global Trust Centre Link Key for participation in a centralised security network
  - Distributed Security Global Link Key for participation in a distributed security network

The pre-configured link keys are defined in the file **bdb_link_keys.c**, from where they are loaded.

Network security is described in Section 2.3.

Parameters

None

Returns

None

BDB_vStart

```
void BDB_vStart(void);
```

Description

This function starts the ZigBee Base Device (ZBD) and must be called after **BDB_vInit()** and just before the application enters the main loop (e.g. **APP_vMainLoop()**).

Depending on the node type and whether the node was previously a member of a network, the function may or may not perform an action. In either case, the function will finally invoke the callback function **APP_vBdbCallback()** with a suitable event.

- If the node was not in a network:
  - For a Router node that supports Touchlink commissioning, the function selects a radio channel for the node from the set of primary channels defined for Touchlink
  - For other Router, Co-ordinator and End Device nodes, no action will be taken and the application will subsequently need to join the node to a network.

  In the above cases, the function will generate the event BDB_EVENT_INIT_SUCCESS.
- If the node was in a network:
  - For Co-ordinator and Router nodes, no action is taken by the function and the event BDB_EVENT_INIT_SUCCESS is generated.
  - For an End Device node, a series of rejoin attempts will be performed. If a rejoin attempt is successful, the event BDB_EVENT_REJOIN_SUCCESS is generated. If all rejoins were unsuccessful, the event BDB_EVENT_REJOIN_FAILURE is generated unless unsecured joins are enabled, in which case a join through Network Steering will be attempted.

The above actions are described in more detail in Section 2.1.

Parameters

None

Returns

None

BDB_eNfStartNwkFormation

```
BDB_teStatus BDB_eNfStartNwkFormation(void);
```

Description

This function starts the Network Formation process and, if required, must be called after **BDB_vStart()**. If it is potentially required on a node, Network Formation must be enabled via the attribute `u8bdbCommissioningMode`.

The function can be called only on a Co-ordinator or Router:

- If called on a Co-ordinator, a centralised security network will be formed
- If called on a Router, a distributed security network will be formed

The above network types are described in Section 2.3.

Once Network Formation has been started, the function will return and the eventual outcome of the Network Formation process will be indicated by an asynchronous event - one of the following:

- BDB_EVENT_NWK_FORMATION_SUCCESS if a centralised or distributed network has been successfully formed
- BDB_EVENT_NWK_FORMATION_FAILURE if a network has not been successfully formed

Network Formation is described in more detail in Section 2.2.3.

Parameters

None

Returns

BDB_E_SUCCESS
(Network Formation has been successfully started) *

BDB_E_ERROR_INVALID_PARAMETER
(End Device has attempted Network Formation)

BDB_E_ERROR_NODE_IS_ON_A_NWK
(node is already in a network)

* The eventual outcome will be indicated by a BDB_EVENT_NWK_FORMATION_SUCCESS or BDB_EVENT_NWK_FORMATION_FAILURE event, as described above.

BDB_eNsStartNwkSteering

```
BDB_teStatus BDB_eNsStartNwkSteering(void);
```

Description

This function starts the Network Steering process and, if required, must be called after **BDB_vStart()**. If it is potentially required on a node, Network Steering must be enabled via the attribute `u8bdbCommissioningMode`.

The actions performed by this function depend on whether the local node is already a member of a network:

- When the node is already in a network and is a Co-ordinator or Router, it opens up the network for other nodes to join. This is for a fixed time-interval of 180 seconds by default, but this interval can be configured (in seconds) using the macro BDBC_MIN_COMMISSIONING_TIME in the **bdb_options.h** file.
- When the node is not already in a network, it searches for a suitable network to join and, if it finds one, attempts to join the network. Once a node has joined the network, the node is authenticated and receives the network key from its parent. If the network has a Trust Centre, the node may then replace its pre-configured link key with one generated and supplied by the Trust Centre.

Once Network Steering has been started, the function will return and the eventual outcome of the Network Steering process will be indicated by an asynchronous event - one of the following:

- BDB_EVENT_NWK_STEERING_SUCCESS if Network Steering has been completed successfully
- BDB_EVENT_NO_NETWORK if no open network was discovered for joining
- BDB_EVENT_NWK_JOIN_FAILURE if the node attempted to join a network but failed

Network Steering is described in more detail in Section 2.2.2.

Parameters

None

Returns

BDB_E_SUCCESS
(Network Steering has been successfully started) *

BDB_E_ERROR_IMPROPER_COMMISSIONING_MODE
(Network Steering is not enabled)

BDB_E_ERROR_COMMISSIONING_IN_PROGRESS
(node is already in a commissioning mode)

BDB_E_ERROR_INVALID_DEVICE
(joining node is a Co-ordinator)

* The eventual outcome will be indicated by a BDB_EVENT_NWK_STEERING_SUCCESS, BDB_EVENT_NO_NETWORK or BDB_EVENT_NWK_JOIN_FAILURE event, as described above.

BDB_eFbTriggerAsInitiator

BDB_teStatus BDB_eFbTriggerAsInitiator(
uint8 *u8SourceEndPointId***);**

Description

This function starts the Finding and Binding process on an initiator endpoint. The function may be called as the result of a user action, such as a button-press. The initiator will remain in Finding and Binding mode for a fixed time-interval (in seconds) defined using the macro BDBC_MIN_COMMISSIONING_TIME in the **bdb_options.h** file.

The initiator node first searches for target endpoints by broadcasting an Identify Query command. If the initiator receives a response from a remote endpoint, it then sends a Simple Descriptor request to this endpoint. If the requested Simple Descriptor is then successfully received back, the initiator checks this descriptor for clusters on the remote endpoint that match its own clusters. It there is at least one matching cluster, the initiator does one of the following:

- If binding is required (indicated by the *u16bdbCommissioningGroupID* attribute being equal to 0xFFFF), the initiator adds the remote endpoint to the local Binding table
- If grouping is required (indicated by the *u16bdbCommissioningGroupID* attribute being equal to a 16-bit group address), the initiator will request that the target endpoint adds the group address to its Group Address table.

Finding and Binding mode is described in Section 2.2.4.

Note: Events are generated during this function call - for details, refer to Section 2.2.4.1.

Parameters

u8SourceEndPointId	Number of initiator endpoint

Returns

BDB_E_SUCCESS
(Finding and Binding has been successfully started)

BDB_E_FAILURE
(invalid endpoint number or unable to broadcast Identify Query command)

BDB_E_ERROR_COMMISSIONING_IN_PROGRESS
(Finding and Binding already on-going)

BDB_vFbExitAsInitiator

```
void BDB_vFbExitAsInitiator(void);
```

Description

This function stops an on-going Finding and Binding process on an initiator endpoint. The function may be called as the result of a user action, such as a button-press or button-release.

Finding and Binding mode is described in Section 2.2.4.

Parameters

None

Returns

None

BDB_eFbTriggerAsTarget

BDB_teStatus BDB_eFbTriggerAsTarget(uint8 *u8EndPoint***);**

Description

This function starts the Finding and Binding process on a target endpoint and must be called locally by the application on the target endpoint. The function may be called as the result of a user action, such as a button-press.

The functions puts the device into identification mode of the Identify cluster for a time-interval (in seconds) which is at least equal to the value defined using the macro BDBC_MIN_COMMISSIONING_TIME in the **bdb_options.h** file. During this time, the target device will generate responses to Identify Query commands, as well as other Finding and Binding commands.

The endpoint can subsequently be brought out of Find and Binding mode locally using the function **BDB_vFbExitAsTarget()** or remotely (by the initiator) using the Identify cluster function **eCLD_IdentifyCommandIdentifyRequestSend()**.

Finding and Binding mode is described in Section 2.2.4.

Parameters

u8EndPoint	Number of target endpoint

Returns

BDB_E_SUCCESS
(Finding and Binding has been successfully started)

BDB_E_FAILURE
(invalid endpoint number or Identify cluster is inaccessible)

BDB_vFbExitAsTarget

void BDB_vFbExitAsTarget(uint8 *u8SourceEndpoint***);**

Description

This function stops an on-going Finding and Binding process on a target endpoint and must be called locally by the application on the target endpoint. The function may be called as the result of a user action, such as a button-press or button-release.

Finding and Binding mode is described in Section 2.2.4.

Parameters

u8SourceEndpoint	Number of target endpoint

Returns

None

BDB_bIsBaseIdle

```
bool_t BDB_bIsBaseIdle(void);
```

Description

This function determines whether the ZigBee Base Device is busy or idle, and therefore whether the node can enter sleep mode. The function returns a Boolean indicating the activity status of the ZigBee Base Device.

If the ZigBee Base Device is idle and the node can go to sleep (indicated by TRUE), it is then the responsibility of the application to put the device into sleep mode.

Parameters

None

Returns

TRUE indicates that the ZigBee Base Device is idle and the node can sleep

FALSE indicates that the ZigBee Base Device is busy

BDB_u8OutOfBandCommissionStartDevice

```
uint8 BDB_u8OutOfBandCommissionStartDevice(
                        BDB_tsOobWriteDataToCommission
*psStartupData);
```

Description

This function can be used to initiate out-of-band commissioning which allows the local device to form a network as a Co-ordinator or to join an existing network as a Router or End Device. The function should be called after **ZPS_eAplAfInit()**. It would be called when commissioning data has been received from another device via out-of-band means. This commissioning data must be supplied to the function in a `BDB_tsOobWriteDataToCommission` structure, described in Section 2.7.5. Not all the data values are mandatory.

The out-of-band commissioning interface makes sensible assumptions about data values and does not allow certain values already in the node to be over-ridden by the commissioning data. For example:

- It will not allow the network address of a Co-ordinator to be set to a non-zero value (since the network address of the Co-ordinator must be zero)
- It will not allow the rejoin flag to be set on a Co-ordinator (since the Co-ordinator cannot leave and then rejoin the network)
- In a centralised network, it will not allow the Trust Centre's IEEE/MAC address to be set to any value other than the Co-ordinator's IEEE/MAC address (since the Co-ordinator is always the Trust Centre)

For an overview of out-of-band commissioning, refer to Section 2.2.5.

Parameters

psStartupData	Pointer to a structure containing commissioning data (see Section 2.7.5)

Returns

BDB_E_SUCCESS
(The device has successfully formed or joined a network)

BDB_E_FAILURE
(The request to form or join a network has not been accepted)

ZPS_NWK_ENUM_INVALID_REQUEST
(The request contained invalid data)

ZPS_APL_APS_E_ILLEGAL_REQUEST
(The stack is not in the correct state to accept the request)

BDB_vOutOfBandCommissionGetData

```
void BDB_vOutOfBandCommissionGetData(
            BDB_tsOobReadDataToAuthenticate
                        *psReturnedCommissioningData);
```

Description

This function can be used to obtain locally stored commissioning data. The obtained data is received in a structure described in Section 2.7.6 and includes the network key. The data can then be passed to higher layers which may encrypt it before sending it by out-of-band means to the other device involved in the commissioning.

A similar set of data but with the network key encrypted can be obtained using the function **BDB_eOutOfBandCommissionGetDataEncrypted()**.

For an overview of out-of-band commissioning, refer to Section 2.2.5.

Parameters

psReturnedCommissioningData	Pointer to a structure to receive the obtained commissioning data (see Section 2.7.6)

Returns

None

BDB_eOutOfBandCommissionGetDataEncrypted

BDB_teStatus
BDB_eOutOfBandCommissionGetDataEncrypted(
    BDB_tsOobWriteDataToAuthenticate **psSrcCredentials*,
    uint8 **pu8ReturnAuthData*,
    uint16 **puSize*);

Description

This function can be used to obtain locally stored commissioning data, including the network key which will be returned encrypted. Authentication data (including an install code) must be provided that will be used to encrypt the network key.

The obtained data is received as a byte stream - the size of the byte stream is also returned. The byte stream contains the following data:

- IEEE/MAC address of the local node as `u64address` (8 bytes)
- Network key encrypted with the data passed via *psSrcCredentials* (16 bytes)
- MIC value generated to validate encryption (4 bytes)
- Key sequence number of active network key (1 byte)
- Active channel number (1 byte)
- PAN ID (2 bytes)
- Extended PAN ID (8 bytes)

The encrypted network key and other obtained data may then be sent by out-of-band means to the other device involved in the commissioning. The receiving device may decrypt the key using the **BDB_bOutOfBandCommissionGetKey()** function.

A similar set of data without encryption of the network key can be obtained using the function **BDB_u8OutOfBandCommissionStartDevice()**.

For an overview of out-of-band commissioning, refer to Section 2.2.5.

Parameters

psSrcCredentials	Pointer to a structure containing authentication data to be used to encrypt the network key (see Section 2.7.7)
pu8ReturnAuthData	Pointer to the start of the returned byte stream containing the obtained data
puSize	Pointer to a location to receive the size of the obtained byte stream

Returns

None

BDB_bOutOfBandCommissionGetKey

```
bool_t BDB_bOutOfBandCommissionGetKey(
                    uint8* pu8InstallCode,
                    uint8* pu8EncKey,
                    uint64 u64ExtAddress,
                    uint8* pu8DecKey,
                    uint8* pu8Mic);
```

Description

This function can be used to decrypt an encrypted security key. It may be used to decrypt the network key received from another device during out-of-band commissioning.

The function requires the install code that was used to generate the pre-configured link key used to encrypt the key.

For an overview of out-of-band commissioning, refer to Section 2.2.5.

Parameters

pu8InstallCode	Pointer to install code used to generate the pre-configured link key used in the encryption
pu8EncKey	Pointer to encrypted key
u64ExtAddress	Pointer to IEEE/MAC address of originating device
pu8DecKey	Pointer to location to receive the decrypted key
pu8Mic	Pointer to the MIC value to be used to validate the decryption

Returns

TRUE if key successfully decrypted, otherwise FALSE

2.7 Structures

2.7.1 BDB_tsBdbEvent

The following structure contains ZigBee Base Device event information that is passed to the **APP_vBdbCallback()** callback function (see Section 2.9).

```
typedef struct
{
    BDB_teBdbEventType   eEventType;
    BDB_tuBdbEventData   uEventData;
} BDB_tsBdbEvent;
```

where:

- `eEventType` is an enumeration indicating the event type - for the possible enumerations, refer to Section 2.9.
- `uEventData` is a union structure containing the event information (if any) - for a description of this structure, refer to Section 2.7.2.

2.7.2 BDB_tuBdbEventData

The following structure is a union containing the data for a ZigBee Base Device event.

```
typedef union
{
    BDB_tsZpsAfEvent             sZpsAfEvent;
    BDB_tsFindAndBindEvent      *psFindAndBindEvent;
} BDB_tuBdbEventData
```

where:

- `sZpsAfEvent` is a structure containing the data for a stack event, indicated by the event type BDB_EVENT_ZPSAF - for a description of this structure, refer to Section 2.7.3.
- `psFindAndBindEvent` is a pointer to a structure containing the data for a 'Finding and Binding' event (see Section 2.9) - for a description of this structure, refer to Section 2.7.4.

2.7.3 BDB_tsZpsAfEvent

The following structure contains the data for a ZigBee stack event (see Section).

```
typedef struct
{
    uint8            u8EndPoint;
    ZPS_tsAfEvent    sStackEvent;
} BDB_tsZpsAfEvent;
```

where:

- `u8EndPoint` is the number of the endpoint on which the event occurred.
- `sStackEvent` is a ZPS structure containing the stack event type and data - this structure is detailed in the *ZigBee 3.0 Stack User Guide (JN-UG-3130)*.

2.7.4 BDB_tsFindAndBindEvent

The following structure contains the data for a 'Finding and Binding' event (see Section 2.9), which is passed to the application during the Finding and Binding process on the initiator.

```
typedef struct{
    uint8                u8InitiatorEp;
    uint8                u8TargetEp;
    uint16               u16TargetAddress;
    uint16               u16ProfileId;
    uint16               u16DeviceId;
    uint8                u8DeviceVersion;
    union {
        uint16           u16ClusterId;
        uint16           u16GroupId;
    }uEvent;
    ZPS_tsAfZdpEvent    *psAfZdpEvent;
    bool                 bAllowBindOrGroup;
    bool                 bGroupCast;
}BDB_tsFindAndBindEvent;
```

where:

- `u8InitiatorEp` is the number of the endpoint involved in the binding/ grouping on the initiator node
- `u8TargetEp` is the number of the endpoint involved in the binding/grouping on the target node
- `u16TargetAddress` is the 16-bit network address of the target node

- `u16ProfileId` is the identifier of the ZigBee application profile supported by the two nodes (for Lighting & Occupancy devices, this is 0x0104)
- `u16DeviceId` is the 16-bit identifier of the ZigBee device type supported by the target endpoints. This must be a device type identifier issued by the ZigBee Alliance.
- `u8DeviceVersion` contains 4 bits (bits 0-3) representing the version of the supported device description on the target node (the default is 0000, unless set to another value according to the application profile used).
- `uEvent` is a union of the following two fields:
  - `u16ClusterId` is the identifier of the cluster involved in the binding
  - `u16GroupId` is the address of the group to which the target endpoint will be assigned
- `psAfZdpEvent` is a pointer to a `ZPS_tsAfZdpEvent` structure containing the generated Finding and Binding event - this ZPS structure is detailed in the *ZigBee 3.0 Stack User Guide (JN-UG-3130)*. The event can be any of the following (detailed in Section 2.9):
  - BDB_EVENT_FB_HANDLE_SIMPLE_DESC_RESP_OF_TARGET
  - BDB_EVENT_FB_CHECK_BEFORE_BINDING_CLUSTER_FOR_TARGET
  - BDB_EVENT_FB_CLUSTER_BIND_CREATED_FOR_TARGET
  - BDB_EVENT_FB_BIND_CREATED_FOR_TARGET
  - BDB_EVENT_FB_GROUP_ADDED_TO_TARGET
  - BDB_EVENT_FB_ERR_BINDING_FAILED
  - BDB_EVENT_FB_ERR_BINDING_TABLE_FULL
  - BDB_EVENT_FB_ERR_GROUPING_FAILED
  - BDB_EVENT_FB_NO_QUERY_RESPONSE
  - BDB_EVENT_FB_TIMEOUT
- `bAllowBindOrGroup` is a Boolean flag that indicates whether the relevant cluster is permitted to participate in a binding or grouping. The default value is TRUE (permitted) but if the application needs to exclude the cluster (and block the binding/grouping) then it should set this field to FALSE.
- `bGroupCast` is a Boolean flag that indicates whether an 'Add Group If Identifying' command should be broadcast to all the identifying targets (TRUE) or an 'Add Group' request should be individually unicast to all the identifying targets. The default value is TRUE.

2.7.5 BDB_tsOobWriteDataToCommission

The following structure contains the data values that are used to initialise a node at the start of out-of-band commissioning of the node.

```
typedef struct {
    uint64    u64PanId;
    uint64    u64TrustCenterAddress;
    uint8*    pu8NwkKey;
    uint8*    pu8InstallCode;
    uint16    u16PanId;
    uint16    u16ShortAddress;
    bool_t    bRejoin;
    uint8     u8ActiveKeySqNum;
    uint8     u8DeviceType;
    uint8     u8RxOnWhenIdle;
    uint8     u8Channel;
    uint8     u8NwkUpdateId;
} BDB_tsOobWriteDataToCommission;
```

where:

- `u64PanId` is the Extended PAN ID of the network to be joined.
- `u64TrustCenterAddress` is the IEEE/MAC address of the Trust Centre in the centralised network to be joined.
- `pu8NwkKey` is a pointer to the network key.
- `pu8InstallCode` is a pointer to an initial link key derived from an install code (see Section 2.3.1).
- `u16PanId` is the PAN ID of the network to be joined.
- `u16ShortAddress` is the network address assigned to the node.
- `bRejoin` is the 'rejoin flag' which indicates whether the node should attempt to rejoin the network if it leaves (TRUE: rejoin, FALSE: do not rejoin).
- `u8ActiveKeySqNum` is the key sequence number associated with the active network key.
- `u8DeviceType` is a value indicating the type of ZigBee node:
  - 0: Co-ordinator
  - 1: Router
  - 2: End Device

  All other values are reserved.

- `u8RxOnWhenIdle` is a value indicating whether the node's receiver is enable during idle periods:
  - 0: Receiver off when idle (sleeping device)
  - 1: Receiver on when idle (non-sleeping device)

  All other values are reserved.
- `u8Channel` is the radio channel number on which the network operates.
- `u8NwkUpdateId` is a unique byte value which is incremented when the network parameters are updated (and is therefore used to determine whether a receiving node has missed an update).

2.7.6 BDB_tsOobReadDataToAuthenticate

The following structure contains data values that are read from the local node during out-of-band commissioning of the node.

```
typedef struct {
    uint8     au8Key[16]__attribute__((aligned (16)));
    uint64    u64TcAddress;
    uint64    u64PanId;
    uint16    u16ShortPanId;
    uint8     u8ActiveKeySeq;
    uint8     u8Channel;
} BDB_tsOobReadDataToAuthenticate;
```

where:

- `au8Key[16]__attribute__((aligned (16)))` is an array containing the current network key, with one byte per array element.
- `u64TcAddress` is the IEEE/MAC address of the Trust Centre of the network to which the node is being commissioned.
- `u64PanId` is the Extended PAN ID of the network to which the node is being commissioned.
- `u16ShortPanId` is the PAN ID of the network to which the node is being commissioned.
- `u8ActiveKeySeq` is the key sequence number of the currently active network key.
- `u8Channel` is the radio channel number on which the network operates.

2.7.7 BDB_tsOobWriteDataToAuthenticate

The following structure contains authentication data that is used to encrypt a security key during out-of-band commissioning of the node.

```
typedef struct {
   uint64    u64ExtAddr;
   uint8*    pu8InstallCode;
} BDB_tsOobWriteDataToAuthenticate;
```

where:

- `u64ExtAddr` is the IEEE/MAC address of the node.
- `pu8InstallCode` is a pointer to a 16-byte install code to be used in the key encryption.

2.8 Enumerations

This section lists and describes the enumerations used on the ZigBee Base Device. However, the ZigBee Base Device event enumerations are detailed in Section 2.9.

2.8.1 BDB_teStatus

The following enumerations are used to indicate the status of certain function calls.

```
typedef enum
{
    BDB_E_SUCCESS,
    BDB_E_FAILURE,
    BDB_E_ERROR_INVALID_PARAMETER,
    BDB_E_ERROR_INVALID_DEVICE,
    BDB_E_ERROR_NODE_IS_ON_A_NWK,
    BDB_E_ERROR_IMPROPER_COMMISSIONING_MODE,
    BDB_E_ERROR_COMMISSIONING_IN_PROGRESS,
}BDB_teStatus;
```

The enumerations are listed and described in the table below.

Enumeration	Description
BDB_E_SUCCESS	Function call was successful in its purpose
BDB_E_FAILURE	Function call failed in its purpose and no other error code is appropriate
BDB_E_ERROR_INVALID_PARAMETER	A specified parameter value was invalid
BDB_E_ERROR_INVALID_DEVICE	Device type is not valid for the operation
BDB_E_ERROR_NODE_IS_ON_A_NWK	Node is already in a network
BDB_E_ERROR_IMPROPER_COMMISSIONING_MODE	The commissioning mode is not appropriate
BDB_E_ERROR_COMMISSIONING_IN_PROGRESS	The commissioning process is in progress

Table 10: Function Status Enumerations

2.8.2 BDB_teCommissioningStatus

The following enumerations are used to indicate the status of the commissioning process for the node.

```
typedef enum
{
    E_BDB_COMMISSIONING_STATUS_SUCCESS,
    E_BDB_COMMISSIONING_STATUS_IN_PROGRESS,
    E_BDB_COMMISSIONING_STATUS_NOT_AA_CAPABLE,
    E_BDB_COMMISSIONING_STATUS_NO_NETWORK,
    E_BDB_COMMISSIONING_STATUS_FORMATION_FAILURE,
    E_BDB_COMMISSIONING_STATUS_NO_IDENTIFY_QUERY_RESPONSE,
    E_BDB_COMMISSIONING_STATUS_BINDING_TABLE_FULL,
    E_BDB_COMMISSIONING_STATUS_NO_SCAN_RESPONSE,
    E_BDB_COMMISSIONING_STATUS_NOT_PERMITTED,
    E_BDB_COMMISSIONING_STATUS_TCLK_EX_FAILURE
}BDB_teCommissioningStatus;
```

The enumerations are listed and described in the table below.

Enumeration	Description
E_BDB_COMMISSIONING_STATUS_SUCCESS	Commissioning has successfully completed
E_BDB_COMMISSIONING_STATUS_IN_PROGRESS	Commissioning is on-going
E_BDB_COMMISSIONING_STATUS_NOT_AA_CAPABLE	Parent cannot assign address to joining node
E_BDB_COMMISSIONING_STATUS_NO_NETWORK	No network was found that can be joined
E_BDB_COMMISSIONING_STATUS_FORMATION_FAILURE	Network formation failed
E_BDB_COMMISSIONING_STATUS_NO_IDENTIFY_QUERY_RESPONSE	No responses were received to an Identify Query command
E_BDB_COMMISSIONING_STATUS_BINDING_TABLE_FULL	The local Binding table is full
E_BDB_COMMISSIONING_STATUS_NO_SCAN_RESPONSE	No responses were received during a channel scan
E_BDB_COMMISSIONING_STATUS_NOT_PERMITTED	Requested commissioning is not permitted
E_BDB_COMMISSIONING_STATUS_TCLK_EX_FAILURE	Trust Centre link key exchange failed

Table 11: Commissioning Status Enumerations

2.9 Events

The ZigBee Base Device has a number of associated events. Some API functions (described in Section 2.6) return immediately and the outcome of the process they invoke is later indicated with the generation of an asynchronous event. A user-defined callback function must be defined in the application to handle these events. The prototype of this callback function is as follows:

void APP_vBdbCallback(BDB_tsBdbEvent ****psBdbEvent*)**

where *psBdbEvent* is a pointer to a `BDB_tsBdbEvent` event structure containing the event information to be passed to the function (for this structure, see Section 2.7.1).

The enumerations for the ZigBee Base Device events are listed below.

```
typedef enum {
    BDB_EVENT_NONE,
    BDB_EVENT_ZPSAF,
    BDB_EVENT_INIT_SUCCESS,
    BDB_EVENT_REJOIN_SUCCESS,
    BDB_EVENT_REJOIN_FAILURE,
    BDB_EVENT_NWK_STEERING_SUCCESS,
    BDB_EVENT_NO_NETWORK,
    BDB_EVENT_NWK_JOIN_SUCCESS,
    BDB_EVENT_NWK_JOIN_FAILURE,
    BDB_EVENT_APP_START_POLLING,
    BDB_EVENT_NWK_FORMATION_SUCCESS,
    BDB_EVENT_NWK_FORMATION_FAILURE,
    BDB_EVENT_FB_HANDLE_SIMPLE_DESC_RESP_OF_TARGET,
    BDB_EVENT_FB_CHECK_BEFORE_BINDING_CLUSTER_FOR_TARGET,
    BDB_EVENT_FB_CLUSTER_BIND_CREATED_FOR_TARGET,
    BDB_EVENT_FB_BIND_CREATED_FOR_TARGET,
    BDB_EVENT_FB_GROUP_ADDED_TO_TARGET,
    BDB_EVENT_FB_ERR_BINDING_FAILED,
    BDB_EVENT_FB_ERR_BINDING_TABLE_FULL,
    BDB_EVENT_FB_ERR_GROUPING_FAILED,
    BDB_EVENT_FB_NO_QUERY_RESPONSE,
    BDB_EVENT_FB_TIMEOUT,
    BDB_EVENT_FB_OVER_AT_TARGET,
    BDB_EVENT_LEAVE_WITHOUT_REJOIN,
} BDB_teBdbEventType;
```

These events are described below.

The events with 'FB' in their names are used in the 'Finding and Binding' process and the event data is contained in the structure `BDB_tsFindAndBindEvent` (see Section 2.7.4).

Note: In addition, certain ZCL events are generated during the Finding and Binding process, and are passed to the callback function **BDB_vZclEventHandler()**, which is supplied with the ZigBee Base Device. For these events, refer to Section 2.2.4.

BDB_EVENT_ZPSAF

This event indicates that a ZigBee stack event has occurred. In this case, the `uEventData` field (of the `BDB_tsBdbEvent` structure) contains a `BDB_tsZpsAfEvent` structure, which itself includes the `ZPS_tsAfEvent` stack event structure.

BDB_EVENT_INIT_SUCCESS

This event is generated when the ZigBee Base Device has been successfully initialised.

BDB_EVENT_REJOIN_SUCCESS

This event is generated when the node has successfully rejoined its previous network.

BDB_EVENT_REJOIN_FAILURE

This event is generated when the node's attempt to rejoin its previous network has failed.

BDB_EVENT_NWK_STEERING_SUCCESS

This event is generated when the Network Steering process has successfully completed and the local node has broadcast either of the following messages:

- Management Permit Joining message to request the network to be opened for other devices to join (this message is broadcast when the local node was already in the network before Network Steering).
- Device Announce message to announce that the local node has just joined the network (this message is broadcast when the local node was not in the network before Network Steering).

BDB_EVENT_NO_NETWORK

This event is generated when no open network open was discovered in a channel scan performed by a device attempting to join a network.

BDB_EVENT_NWK_JOIN_SUCCESS

This event is generated when the node has successfully joined a network.

BDB_EVENT_NWK_JOIN_FAILURE

This event is generated when the node attempted to a join a network but failed.

BDB_EVENT_APP_START_POLLING

This event is generated on an End Device during the Trust Centre link key exchange procedure to instruct the application to start fast polling of its parent, in order to retrieve packets received as part of the exchange procedure.

BDB_EVENT_NWK_FORMATION_SUCCESS

This event is generated at the end of the Network Formation process when a centralised or distributed has been successfully formed by the local node.

BDB_EVENT_NWK_FORMATION_FAILURE

This event is generated at the end of the Network Formation process if the local node failed to form a network.

BDB_EVENT_FB_HANDLE_SIMPLE_DESC_RESP_OF_TARGET

This event indicates that the initiator has received a Simple Descriptor response from a target. This event can be used by the application to determine which type of device (e.g. Dimmable Light, On/Off Light) the initiator is binding to. The information provided to the application is:

- `u8InitiatorEp`
- `u8TargetEp`
- `u16TargetAddress`
- `u16ProfileId`
- `u16DeviceId`
- `u8DeviceVersion`
- `psAfZdpEvent` (points to received Simple Descriptor)

BDB_EVENT_FB_CHECK_BEFORE_BINDING_CLUSTER_FOR_TARGET

This event is generated just before creating a Binding table entry for a cluster. It gives the application an opportunity to exclude clusters from binding by setting the `bAllowBindOrGroup` flag to FALSE (by default it is TRUE). This event can also be used when the application needs to perform a group binding by setting the attribute *u16bdbCommissioningGroupID* to a value other than 0xFFFF. Moreover, this event also allows the application to decide whether to broadcast an 'Add Group If Identifying' command to all the identifying targets by setting `bGroupCast` to TRUE (by default it is assumed to be FALSE) or unicast an 'Add Group' request individually to all the identifying targets. The information provided to the application is:

- `u8InitiatorEp`
- `u8TargetEp`
- `u16TargetAddress`
- `u16ClusterId`
- `bAllowBindOrGroup`
- `bGroupCast`
- `psAfZdpEvent` (points to received Simple Descriptor)

BDB_EVENT_FB_CLUSTER_BIND_CREATED_FOR_TARGET

This event is generated per cluster for every binding or grouping created. The event may be generated more than once for the same target device. For example, when binding a Colour Dimmer Switch to a Dimmable Light, the event will be generated twice: once for the On/Off cluster and once for the Level Control Cluster. The information provided to the application is:

- `u8InitiatorEp`
- `u8TargetEp`
- `u16TargetAddress`
- `u16ClusterId`

BDB_EVENT_FB_BIND_CREATED_FOR_TARGET

This event is generated once all address bindings have been completed. The application can then send a ‘Stop Identifying’ command to the bound target. The information provided to the application is:

- `u8InitiatorEp`
- `u8TargetEp`
- `u16TargetAddress`

BDB_EVENT_FB_GROUP_ADDED_TO_TARGET

This event is generated once the 'Add Group' or 'Add Group If Identifying' has been sent, in order to inform the application that grouping has been completed from the initiator's perspective. The application can then groupcast a 'Stop Identifying' command to the grouped targets. The information provided to the application is:

- `u8InitiatorEp`
- `u8TargetEp`
- `u16GroupId`
- `u16TargetAddress`
- `psAfZdpEvent`

BDB_EVENT_FB_ERR_BINDING_FAILED

This event is generated to indicate that an unexpected error has occurred while creating a Binding table entry.

BDB_EVENT_FB_ERR_BINDING_TABLE_FULL

This event is generated to inform the application that the Binding table is full and therefore the Finding and Binding process has failed. As a result, the ZigBee Base Device will exit the Finding and Binding process.

BDB_EVENT_FB_ERR_GROUPING_FAILED

This event is generated to indicate that a grouping has failed, since the initiator was not able to send an 'Add Group' or 'Add Group If Identifying' request.

BDB_EVENT_FB_NO_QUERY_RESPONSE

This event indicates that the initiator did not receive an Identify Query response within BDB_FB_RESEND_IDENTIFY_QUERY_TIME (default value is 10) seconds. The information provided to the application is:

- `u8InitiatorEp`

BDB_EVENT_FB_TIMEOUT

This event indicates that the commissioning timer expired after a period defined by the constant BDBC_MIN_COMMISSIONING_TIME (180 seconds by default). The information provided to the application is:

- `u8InitiatorEp`

BDB_EVENT_FB_OVER_AT_TARGET

This event indicates that the Finding and Binding process has ended on the target node because the identify time reached zero or a remote node forced it to go to zero.

BDB_EVENT_LEAVE_WITHOUT_REJOIN

This event is generated when the node has been instructed to leave the network without subsequently attempting to rejoin the network.

2.10 Compile-time Options

Compile-time options can be configured through definitions in the file **bdb_options.h**. This allows custom values to be defined for ZigBee Base Device attributes and constants. If the value of an attribute or constant is not defined in this file, the default value for the attribute or constant will be used.

Attributes

The following macros can be used to pre-configure values for the ZigBee Base Device attributes (listed and described in Section 2.5.1):

- BDB_COMMISSIONING_GROUP_ID
- BDB_COMMISSIONING_MODE
- BDB_COMMISSIONING_STATUS
- BDB_JOINING_NODE_EUI64
- BDB_JOIN_USES_INSTALL_CODE_KEY
- BDB_NODE_JOIN_LINK_KEY_TYPE
- BDB_PRIMARY_CHANNEL_SET
- BDB_SCAN_DURATION
- BDB_SECONDARY_CHANNEL_SET
- BDB_TC_LINK_KEY_EXCHANGE_ATTEMPTS
- BDB_TC_LINK_KEY_EXCHANGE_ATTEMPTS_MAX
- BDB_TC_LINK_KEY_EXCHANGE_METHOD
- BDB_TRUST_CENTER_NODE_JOIN_TIMEOUT
- BDB_TRUST_CENTER_REQUIRE_KEYEXCHANGE

For example, to set the maximum number of key establishment attempts to 5, include the following line:

```
#define BDB_TC_LINK_KEY_EXCHANGE_ATTEMPTS_MAX      5
```

Constants

The following macros can be used to set values for the ZigBee Base Device constants (listed and described in Section 2.5.2):

- BDBC_MAX_SAME_NETWORK_RETRY_ATTEMPTS
- BDBC_MIN_COMMISSIONING_TIME
- BDBC_REC_SAME_NETWORK_RETRY_ATTEMPTS
- BDBC_TC_LINK_KEY_EXCHANGE_TIMEOUT
- BDBC_TL_INTERPAN_TRANS_ID_LIFETIME
- BDBC_TL_MIN_STARTUP_DELAY_TIME
- BDBC_TL_PRIMARY_CHANNEL_SET
- BDBC_TL_RX_WINDOW_DURATION
- BDBC_TL_SCAN_TIME_BASE_DURATION_MS
- BDBC_TL_SECONDARY_CHANNEL_SET

For example, to set the minimum commissioning time for which a network will be open to joining to 240 seconds, include the following line:

```
#define BDBC_MIN_COMMISSIONING_TIME    240
```

(this minimum commissioning time should set to a value below 255 seconds)

3. Lighting and Occupancy Device Types

This chapter details the ZigBee device types that are collected together in the *ZigBee Lighting and Occupancy Device Specification (15-0014-01)* from the ZigBee Alliance.

Note: Lighting and Occupancy is not an application profile but devices in this collection use the application profile identifier 0x0104 that was previously used for the Home Automation application profile. This ensures backward compatibility with applications for devices based on the Home Automation 1.2 profile.

The ZigBee Lighting and Occupancy (ZLO) device types are listed below:

Device Type	Device ID	Reference
On/Off Light	0x0100	Section 3.1
Dimmable Light	0x0101	Section 3.2
Colour Dimmable Light	0x0102	Section 3.3
On/Off Light Switch	0x0103	Section 3.4
Dimmer Switch	0x0104	Section 3.5
Colour Dimmer Switch	0x0105	Section 3.6
Light Sensor	0x0106	Section 3.7
Occupancy Sensor	0x0107	Section 3.8
On/Off Ballast	0x0108	Section 3.9
Dimmable Ballast	0x0109	Section 3.10
On/Off Plug-in Unit	0x010A	Section 3.11
Dimmable Plug-in Unit	0x010B	Section 3.12
Colour Temperature Light	0x010C	Section 3.13
Extended Colour Light	0x010D	Section 3.14
Light Level Sensor	0x010E	Section 3.15
Colour Controller	0x0800	Section 3.16
Colour Scene Controller	0x0810	Section 3.17
Non-Colour Controller	0x0820	Section 3.18
Non-Colour Scene Controller	0x0830	Section 3.19
Control Bridge	0x0840	Section 3.20
On/Off Sensor	0x0850	Section 3.21

Table 1: Lighting and Occupancy Device Types

3.1 On/Off Light

The On/Off Light device is simply a light that can be switched on and off (two states only and no intermediate levels).

- The Device ID is 0x0100
- The header file for the device is **on_off_light.h**
- The clusters supported by the device are listed in Section 3.1.1
- The device structure, `tsZLO_OnOffLightDevice`, is listed in Section 3.1.2
- The endpoint registration function for the device, **eZLO_RegisterOnOffLightEndPoint()**, is detailed in Section 3.1.3

3.1.1 Supported Clusters

The clusters used by the On/Off Light device are listed in the table below.

Server (Input) Side	Client (Output) Side
Mandatory	
Basic	
Identify	
On/Off	
Scenes	
Groups	
Optional	
Level Control	OTA Upgrade
Touchlink Commissioning	Occupancy Sensing

Table 2: Clusters for On/Off Light

The mandatory attributes within each cluster for this device type are indicated in the *ZigBee Lighting and Occupancy Device Specification (15-0014-01)*.

3.1.2 Device Structure

The following `tsZLO_OnOffLightDevice` structure is the shared structure for an On/Off Light device:

```
typedef struct
{
    tsZCL_EndPointDefinition sEndPoint;

    /* Cluster instances */
    tsZLO_OnOffLightDeviceClusterInstances sClusterInstance;

    #if (defined CLD_BASIC) && (defined BASIC_SERVER)
        /* Basic Cluster - Server */
        tsCLD_Basic sBasicServerCluster;
    #endif

    #if (defined CLD_ONOFF) && (defined ONOFF_SERVER)
        /* On/Off Cluster - Server */
        tsCLD_OnOff sOnOffServerCluster;
        tsCLD_OnOffCustomDataStructure sOnOffServerCustomDataStructure;
    #endif

    #if (defined CLD_GROUPS) && (defined GROUPS_SERVER)
        /* Groups Cluster - Server */
        tsCLD_Groups sGroupsServerCluster;
        tsCLD_GroupsCustomDataStructure sGroupsServerCustomDataStructure;
    #endif

    #if (defined CLD_SCENES) && (defined SCENES_SERVER)
        /* Scenes Cluster - Server */
        tsCLD_Scenes sScenesServerCluster;
        tsCLD_ScenesCustomDataStructure sScenesServerCustomDataStructure;
    #endif

    #if (defined CLD_IDENTIFY) && (defined IDENTIFY_SERVER)
        /* Identify Cluster - Server */
        tsCLD_Identify sIdentifyServerCluster;
        tsCLD_IdentifyCustomDataStructure
                                sIdentifyServerCustomDataStructure;
    #endif

    /* On Off light device 2 optional clusters for the server */

    #if (defined CLD_LEVEL_CONTROL) && (defined LEVEL_CONTROL_SERVER)
        /* LevelControl Cluster - Server */
        tsCLD_LevelControl sLevelControlServerCluster;
        tsCLD_LevelControlCustomDataStructure
                                sLevelControlServerCustomDataStructure;
    #endif
    #if (defined CLD_ZLL_COMMISSION) && (defined ZLL_COMMISSION_SERVER)
        tsCLD_ZllCommission sZllCommissionServerCluster;
        tsCLD_ZllCommissionCustomDataStructure
                                sZllCommissionServerCustomDataStructure;
    #endif

    /* On Off light device 2 optional clusters for the client */

    #if (defined CLD_OTA) && (defined OTA_CLIENT)
        /* OTA cluster - Client */
        tsCLD_AS_Ota sCLD_OTA;
        tsOTA_Common sCLD_OTA_CustomDataStruct;
    #endif

    #if (defined CLD_OCCUPANCY_SENSING) && (defined 
OCCUPANCY_SENSING_CLIENT)
        /* Occupancy Sensing Cluster - Client */
        tsCLD_OccupancySensing sOccupancySensingClientCluster;
    #endif

} tsZLO_OnOffLightDevice;
```

3.1.3 Registration Function

The following **eZLO_RegisterOnOffLightEndPoint()** function is the endpoint registration function for an On/Off Light device.

teZCL_Status eZLO_RegisterOnOffLightEndPoint(
uint8 *u8EndPointIdentifier*,
tfpZCL_ZCLCallBackFunction *cbCallBack*,
tsZLO_OnOffLightDevice **psDeviceInfo*);

Description

This function is used to register an endpoint which will support an On/Off Light device. The function must be called after **eZCL_Initialise()**.

The specified identifier for the endpoint is a number in the range 1 to 240 (endpoint 0 is reserved for ZigBee use). Application endpoints are normally numbered consecutively starting at 1. The specified number must be less than or equal to the value of ZLO_NUMBER_OF_ENDPOINTS defined in the **zcl_options.h** file, which represents the highest endpoint number used for applications.

As part of this function call, you must specify a user-defined callback function that will be invoked when an event occurs that is associated with the endpoint. This callback function is defined according to the typedef:

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

You must also provide a pointer to a `tsZLO_OnOffLightDevice` structure (see Section 3.1.2) which will be used to store all variables relating to the On/Off Light device associated with the endpoint. The `sEndPoint` and `sClusterInstance` fields of this structure are set by this function and must not be directly written to by the application.

The function may be called multiple times if more than one endpoint is being used - for example, if more than one On/Off Light device is housed in the same hardware, sharing the same module.

Parameters

u8EndPointIdentifier	Endpoint that is to be associated with the registered structure and callback function
cbCallBack	Pointer to the callback function that will be used to indicate events to the application for this endpoint
psDeviceInfo	Pointer to the structure that will act as storage for all variables related to the device being registered on this endpoint (see Section 3.1.2). The `sEndPoint` and `sClusterInstance` fields are set by this register function for internal use and must not be written to by the application

Returns

E_ZCL_SUCCESS
E_ZCL_FAIL
E_ZCL_ERR_PARAMETER_NULL
E_ZCL_ERR_PARAMETER_RANGE
E_ZCL_ERR_EP_RANGE
E_ZCL_ERR_CLUSTER_0
E_ZCL_ERR_CALLBACK_NULL
E_ZCL_ERR_CLUSTER_NULL
E_ZCL_ERR_SECURITY_RANGE
E_ZCL_ERR_CLUSTER_ID_RANGE
E_ZCL_ERR_MANUFACTURER_SPECIFIC
E_ZCL_ERR_ATTRIBUTE_TYPE_UNSUPPORTED
E_ZCL_ERR_ATTRIBUTE_ID_ORDER
E_ZCL_ERR_ATTRIBUTES_ACCESS

The above codes are described in the *ZCL User Guide (JN-UG-3132)*.

3.2 Dimmable Light

The Dimmable Light device is a light that can have its luminance varied, and can be switched on and off. The permitted range of light levels is 0x01 to 0xFE.

- The Device ID is 0x0101
- The header file for the device is **dimmable_light.h**
- The clusters supported by the device are listed in Section 3.2.1
- The device structure, `tsZLO_DimmableLightDevice`, is listed in Section 3.2.2
- The endpoint registration function for the device, **eZLO_RegisterDimmableLightEndPoint()**, is detailed in Section 3.2.3

3.2.1 Supported Clusters

The clusters used by the Dimmable Light device are listed in the table below.

Server (Input) Side	Client (Output) Side
Mandatory	
Basic	
Identify	
On/Off	
Level Control	
Scenes	
Groups	
Optional	
Touchlink Commissioning	OTA Upgrade
	Occupancy Sensing

Table 3: Clusters for Dimmable Light

The mandatory attributes within each cluster for this device type are indicated in the *ZigBee Lighting and Occupancy Device Specification (15-0014-01)*.

3.2.2 Device Structure

The following `tsZLO_DimmableLightDevice` structure is the shared structure for a Dimmable Light device:

```
typedef struct
{
    tsZCL_EndPointDefinition sEndPoint;

    /* Cluster instances */
    tsZLO_DimmableLightDeviceClusterInstances sClusterInstance;

    #if (defined CLD_BASIC) && (defined BASIC_SERVER)
        /* Basic Cluster - Server */
        tsCLD_Basic sBasicServerCluster;
    #endif

    #if (defined CLD_IDENTIFY) && (defined IDENTIFY_SERVER)
        /* Identify Cluster - Server */
        tsCLD_Identify sIdentifyServerCluster;
        tsCLD_IdentifyCustomDataStructure
                            sIdentifyServerCustomDataStructure;
    #endif

    #if (defined CLD_ONOFF) && (defined ONOFF_SERVER)
        /* On/Off Cluster - Server */
        tsCLD_OnOff sOnOffServerCluster;
        tsCLD_OnOffCustomDataStructure sOnOffServerCustomDataStructure;
    #endif

    #if (defined CLD_GROUPS) && (defined GROUPS_SERVER)
        /* Groups Cluster - Server */
        tsCLD_Groups sGroupsServerCluster;
        tsCLD_GroupsCustomDataStructure sGroupsServerCustomDataStructure;
    #endif

    #if (defined CLD_SCENES) && (defined SCENES_SERVER)
        /* Scenes Cluster - Server */
        tsCLD_Scenes sScenesServerCluster;
        tsCLD_ScenesCustomDataStructure sScenesServerCustomDataStructure;
    #endif

    #if (defined CLD_LEVEL_CONTROL) && (defined LEVEL_CONTROL_SERVER)
        /* LevelControl Cluster - Server */
        tsCLD_LevelControl sLevelControlServerCluster;
        tsCLD_LevelControlCustomDataStructure
                              sLevelControlServerCustomDataStructure;
    #endif

    #if (defined CLD_ZLL_COMMISSION) && (defined ZLL_COMMISSION_SERVER)
```

```
        tsCLD_ZllCommission sZllCommissionServerCluster;
        tsCLD_ZllCommissionCustomDataStructure
                              sZllCommissionServerCustomDataStructure;
    #endif

    #if (defined CLD_OTA) && (defined OTA_CLIENT)
        /* OTA cluster - Client */
        tsCLD_AS_Ota sCLD_OTA;
        tsOTA_Common sCLD_OTA_CustomDataStruct;
    #endif

    #if (defined CLD_OCCUPANCY_SENSING) && (defined
OCCUPANCY_SENSING_CLIENT)
        /* Occupancy Sensing Cluster - Client */
        tsCLD_OccupancySensing sOccupancySensingClientCluster;
    #endif

} tsZLO_DimmableLightDevice;
```

3.2.3 Registration Function

The following **eZLO_RegisterDimmableLightEndPoint()** function is the endpoint registration function for a Dimmable Light device.

teZCL_Status eZLO_RegisterDimmableLightEndPoint(
uint8 *u8EndPointIdentifier*,
tfpZCL_ZCLCallBackFunction *cbCallBack*,
tsZLO_DimmableLightDevice **psDeviceInfo*);

Description

This function is used to register an endpoint which will support a Dimmable Light device. The function must be called after **eZCL_Initialise()**.

The specified identifier for the endpoint is a number in the range 1 to 240 (endpoint 0 is reserved for ZigBee use). Application endpoints are normally numbered consecutively starting at 1. The specified number must be less than or equal to the value of ZLO_NUMBER_OF_ENDPOINTS defined in the **zcl_options.h** file, which represents the highest endpoint number used for applications.

As part of this function call, you must specify a user-defined callback function that will be invoked when an event occurs that is associated with the endpoint. This callback function is defined according to the typedef:

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

You must also provide a pointer to a `tsZLO_DimmableLightDevice` structure (see Section 3.2.2) which will be used to store all variables relating to the Dimmable Light device associated with the endpoint. The `sEndPoint` and `sClusterInstance` fields of this structure are set by this function and must not be directly written to by the application.

The function may be called multiple times if more than one endpoint is being used - for example, if more than one Dimmable Light device is housed in the same hardware, sharing the same module.

Parameters

u8EndPointIdentifier	Endpoint that is to be associated with the registered structure and callback function
cbCallBack	Pointer to the function that will be used to indicate events to the application for this endpoint
psDeviceInfo	Pointer to the structure that will act as storage for all variables related to the device being registered on this endpoint (see Section 3.2.2). The `sEndPoint` and `sClusterInstance` fields are set by this register function for internal use and must not be written to by the application

Returns

E_ZCL_SUCCESS
E_ZCL_FAIL
E_ZCL_ERR_PARAMETER_NULL
E_ZCL_ERR_PARAMETER_RANGE
E_ZCL_ERR_EP_RANGE
E_ZCL_ERR_CLUSTER_0
E_ZCL_ERR_CALLBACK_NULL
E_ZCL_ERR_CLUSTER_NULL
E_ZCL_ERR_SECURITY_RANGE
E_ZCL_ERR_CLUSTER_ID_RANGE
E_ZCL_ERR_MANUFACTURER_SPECIFIC
E_ZCL_ERR_ATTRIBUTE_TYPE_UNSUPPORTED
E_ZCL_ERR_ATTRIBUTE_ID_ORDER
E_ZCL_ERR_ATTRIBUTES_ACCESS

The above codes are described in the *ZCL User Guide (JN-UG-3132)*.

3.3 Colour Dimmable Light

The Colour Dimmable Light device is a multi-colour light that can have its hue, saturation and luminance varied, and can be switched on and off.

- The Device ID is 0x0102
- The header file for the device is **colour_dimmable_light.h**
- The clusters supported by the device are listed in Section 3.3.1
- The device structure, `tsZLO_ColourDimmableLightDevice`, is listed in Section 3.3.2
- The endpoint registration function for the device, **eZLO_RegisterColourDimmableLightEndPoint()**, is detailed in Section 3.3.3

3.3.1 Supported Clusters

The clusters used by the Colour Dimmable Light device are listed in the table below.

Server (Input) Side	Client (Output) Side
Mandatory	
Basic	
Identify	
On/Off	
Level Control	
Colour Control	
Scenes	
Groups	
Optional	
Touchlink Commissioning	OTA Upgrade
	Occupancy Sensing

Table 4: Clusters for Colour Dimmable Light

The mandatory attributes within each cluster for this device type are indicated in the *ZigBee Lighting and Occupancy Device Specification (15-0014-01)*.

3.3.2 Device Structure

The following `tsZLO_ColourDimmableLightDevice` structure is the shared structure for a Colour Dimmable Light device:

```
typedef struct
{
    tsZCL_EndPointDefinition sEndPoint;

    /* Cluster instances */
    tsZLO_ColourDimmableLightDeviceClusterInstances sClusterInstance;

    #if (defined CLD_BASIC) && (defined BASIC_SERVER)
        /* Basic Cluster - Server */
        tsCLD_Basic sBasicServerCluster;
    #endif

    #if (defined CLD_IDENTIFY) && (defined IDENTIFY_SERVER)
        /* Identify Cluster - Server */
        tsCLD_Identify sIdentifyServerCluster;
        tsCLD_IdentifyCustomDataStructure
                            sIdentifyServerCustomDataStructure;
    #endif

    #if (defined CLD_ONOFF) && (defined ONOFF_SERVER)
        /* On/Off Cluster - Server */
        tsCLD_OnOff sOnOffServerCluster;
        tsCLD_OnOffCustomDataStructure sOnOffServerCustomDataStructure;
    #endif

    #if (defined CLD_GROUPS) && (defined GROUPS_SERVER)
        /* Groups Cluster - Server */
        tsCLD_Groups sGroupsServerCluster;
        tsCLD_GroupsCustomDataStructure sGroupsServerCustomDataStructure;
    #endif

    #if (defined CLD_SCENES) && (defined SCENES_SERVER)
        /* Scenes Cluster - Server */
        tsCLD_Scenes sScenesServerCluster;
        tsCLD_ScenesCustomDataStructure sScenesServerCustomDataStructure;
    #endif

    #if (defined CLD_LEVEL_CONTROL) && (defined LEVEL_CONTROL_SERVER)
        /* LevelControl Cluster - Server */
        tsCLD_LevelControl sLevelControlServerCluster;
        tsCLD_LevelControlCustomDataStructure
                          sLevelControlServerCustomDataStructure;
    #endif

    #if (defined CLD_COLOUR_CONTROL) && (defined COLOUR_CONTROL_SERVER)
```

```
        /* Colour Control Cluster - Server */
        tsCLD_ColourControl sColourControlServerCluster;
        tsCLD_ColourControlCustomDataStructure
                          sColourControlServerCustomDataStructure;
    #endif

    #if (defined CLD_ZLL_COMMISSION) && (defined ZLL_COMMISSION_SERVER)
      tsCLD_ZllCommission                    sZllCommissionServerCluster;
        tsCLD_ZllCommissionCustomDataStructure
                          sZllCommissionServerCustomDataStructure;
    #endif

    #if (defined CLD_OTA) && (defined OTA_CLIENT)
        /* OTA cluster - Client */
        tsCLD_AS_Ota sCLD_OTA;
        tsOTA_Common sCLD_OTA_CustomDataStruct;
    #endif

    #if (defined CLD_OCCUPANCY_SENSING) && (defined
OCCUPANCY_SENSING_CLIENT)
        /* Occupancy Sensing Cluster - Client */
        tsCLD_OccupancySensing sOccupancySensingClientCluster;
    #endif
} tsZLO_ColourDimmableLightDevice;
```

3.3.3 Registration Function

The following **eZLO_RegisterColourDimmableLightEndPoint()** function is the endpoint registration function for a Colour Dimmable Light device.

teZCL_Status
eZLO_RegisterColourDimmableLightEndPoint(
    uint8 *u8EndPointIdentifier*,
    tfpZCL_ZCLCallBackFunction *cbCallBack*,
    tsZLO_ColourDimmableLightDevice **psDeviceInfo*);

Description

This function is used to register an endpoint which will support a Colour Dimmable Light device. The function must be called after **eZCL_Initialise()**.

The specified identifier for the endpoint is a number in the range 1 to 240 (endpoint 0 is reserved for ZigBee use). Application endpoints are normally numbered consecutively starting at 1. The specified number must be less than or equal to the value of ZLO_NUMBER_OF_ENDPOINTS defined in the **zcl_options.h** file, which represents the highest endpoint number used for applications.

As part of this function call, you must specify a user-defined callback function that will be invoked when an event occurs that is associated with the endpoint. This callback function is defined according to the typedef:

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

You must also provide a pointer to a `tsZLO_ColourDimmableLightDevice` structure (see Section 3.3.2) which will be used to store all variables relating to the Colour Dimmable Light device associated with the endpoint. The `sEndPoint` and `sClusterInstance` fields of this structure are set by this function and must not be directly written to by the application.

The function may be called multiple times if more than one endpoint is being used - for example, if more than one Colour Dimmable Light device is housed in the same hardware, sharing the same module.

Parameters

u8EndPointIdentifier	Endpoint that is to be associated with the registered structure and callback function
cbCallBack	Pointer to the function that will be used to indicate events to the application for this endpoint
psDeviceInfo	Pointer to the structure that will act as storage for all variables related to the device being registered on this endpoint (see Section 3.3.2). The `sEndPoint` and `sClusterInstance` fields are set by this register function for internal use and must not be written to by the application

Returns

E_ZCL_SUCCESS
E_ZCL_FAIL
E_ZCL_ERR_PARAMETER_NULL
E_ZCL_ERR_PARAMETER_RANGE
E_ZCL_ERR_EP_RANGE
E_ZCL_ERR_CLUSTER_0
E_ZCL_ERR_CALLBACK_NULL
E_ZCL_ERR_CLUSTER_NULL
E_ZCL_ERR_SECURITY_RANGE
E_ZCL_ERR_CLUSTER_ID_RANGE
E_ZCL_ERR_MANUFACTURER_SPECIFIC
E_ZCL_ERR_ATTRIBUTE_TYPE_UNSUPPORTED
E_ZCL_ERR_ATTRIBUTE_ID_ORDER
E_ZCL_ERR_ATTRIBUTES_ACCESS

The above codes are described in the *ZCL User Guide (JN-UG-3132)*.

3.4 On/Off Light Switch

The On/Off Light Switch device is used to switch a light device on and off by sending on, off and toggle commands to the target device.

- The Device ID is 0x0103
- The header file for the device is **on_off_light_switch.h**
- The clusters supported by the device are listed in Section 3.4.1
- The device structure, `tsZLO_OnOffLightSwitchDevice`, is listed in Section 3.4.2
- The endpoint registration function for the device, **eZLO_RegisterOnOffLightSwitchEndPoint()**, is detailed in Section 3.4.3

3.4.1 Supported Clusters

The clusters used by the On/Off Light Switch device are listed in the table below.

Server (Input) Side	**Client (Output) Side**
Mandatory	
Basic	On/Off
Identify	Identify
Optional	
On/Off Switch Configuration	OTA Upgrade
	Scenes
	Groups

Table 5: Clusters for On/Off Light Switch

The mandatory attributes within each cluster for this device type are indicated in the *ZigBee Lighting and Occupancy Device Specification (15-0014-01)*.

3.4.2 Device Structure

The following `tsZLO_OnOffLightSwitchDevice` structure is the shared structure for an On/Off Light Switch device:

```
typedef struct
{
    tsZCL_EndPointDefinition sEndPoint;

    /* Cluster instances */
    tsZLO_OnOffLightSwitchDeviceClusterInstances sClusterInstance;

    /* Mandatory server clusters */
    #if (defined CLD_BASIC) && (defined BASIC_SERVER)
        /* Basic Cluster - Server */
        tsCLD_Basic sBasicServerCluster;
    #endif

    #if (defined CLD_IDENTIFY) && (defined IDENTIFY_SERVER)
        /* Identify Cluster - Server */
        tsCLD_Identify sIdentifyServerCluster;
        tsCLD_IdentifyCustomDataStructure
                       sIdentifyServerCustomDataStructure;
    #endif

    /* Recommended Optional server */
    #if (defined CLD_OOSC) && (defined OOSC_SERVER)
        /* On/Off Switch Configuration Cluster - Server */
        tsCLD_OOSC sOOSCServerCluster;
    #endif

    /* Mandatory client clusters */
    #if (defined CLD_ONOFF) && (defined ONOFF_CLIENT)
        tsCLD_OnOff sOnOffClientCluster;
    #endif

    #if (defined CLD_IDENTIFY) && (defined IDENTIFY_CLIENT)
        /* Identify Cluster - Client */
        tsCLD_Identify sIdentifyClientCluster;
        tsCLD_IdentifyCustomDataStructure
                       sIdentifyClientCustomDataStructure;
    #endif

    #if (defined CLD_BASIC) && (defined BASIC_CLIENT)
        /* Basic Cluster - Client */
        tsCLD_Basic sBasicClientCluster;
    #endif

    /* Recommended Optional client clusters */
    #if (defined CLD_SCENES) && (defined SCENES_CLIENT)
```

```
        /* Scenes Cluster - Client */
        tsCLD_Scenes sScenesClientCluster;
        tsCLD_ScenesCustomDataStructure sScenesClientCustomDataStructure;
    #endif

    #if (defined CLD_GROUPS) && (defined GROUPS_CLIENT)
        /* Groups Cluster - Client */
        tsCLD_Groups sGroupsClientCluster;
        tsCLD_GroupsCustomDataStructure sGroupsClientCustomDataStructure;
    #endif

    #if (defined CLD_OTA) && (defined OTA_CLIENT)
        tsCLD_AS_Ota sCLD_OTA;
        tsOTA_Common sCLD_OTA_CustomDataStruct;
    #endif
} tsZLO_OnOffLightSwitchDevice;
```

3.4.3 Registration Function

The following **eZLO_RegisterOnOffLightSwitchEndPoint()** function is the endpoint registration function for an On/Off Light Switch device.

teZCL_Status eZLO_RegisterOnOffLightSwitchEndPoint(
uint8 *u8EndPointIdentifier*,
tfpZCL_ZCLCallBackFunction *cbCallBack*,
tsZLO_OnOffLightSwitchDevice **psDeviceInfo*);

Description

This function is used to register an endpoint which will support an On/Off Light Switch device. The function must be called after **eZCL_Initialise()**.

The specified identifier for the endpoint is a number in the range 1 to 240 (endpoint 0 is reserved for ZigBee use). Application endpoints are normally numbered consecutively starting at 1. The specified number must be less than or equal to the value of ZLO_NUMBER_OF_ENDPOINTS defined in the **zcl_options.h** file, which represents the highest endpoint number used for applications.

As part of this function call, you must specify a user-defined callback function that will be invoked when an event occurs that is associated with the endpoint. This callback function is defined according to the typedef:

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

You must also provide a pointer to a `tsZLO_OnOffLightSwitchDevice` structure (see Section 3.4.2) which will be used to store all variables relating to the On/Off Light Switch device associated with the endpoint. The `sEndPoint` and `sClusterInstance` fields of this structure are set by this function and must not be directly written to by the application.

The function may be called multiple times if more than one endpoint is being used - for example, if more than one On/Off Light Switch device is housed in the same hardware, sharing the same module.

Parameters

u8EndPointIdentifier	Endpoint that is to be associated with the registered structure and callback function
cbCallBack	Pointer to the function that will be used to indicate events to the application for this endpoint
psDeviceInfo	Pointer to the structure that will act as storage for all variables related to the device being registered on this endpoint (see Section 3.4.2). The `sEndPoint` and `sClusterInstance` fields are set by this register function for internal use and must not be written to by the application

Returns

E_ZCL_SUCCESS
E_ZCL_FAIL
E_ZCL_ERR_PARAMETER_NULL
E_ZCL_ERR_PARAMETER_RANGE
E_ZCL_ERR_EP_RANGE
E_ZCL_ERR_CLUSTER_0
E_ZCL_ERR_CALLBACK_NULL
E_ZCL_ERR_CLUSTER_NULL
E_ZCL_ERR_SECURITY_RANGE
E_ZCL_ERR_CLUSTER_ID_RANGE
E_ZCL_ERR_MANUFACTURER_SPECIFIC
E_ZCL_ERR_ATTRIBUTE_TYPE_UNSUPPORTED
E_ZCL_ERR_ATTRIBUTE_ID_ORDER
E_ZCL_ERR_ATTRIBUTES_ACCESS

The above codes are described in the *ZCL User Guide (JN-UG-3132)*.

3.5 Dimmer Switch

The Dimmer Switch device is used to control a characteristic of a light (e.g. luminance) and to switch the light device on and off.

- The Device ID is 0x0104
- The header file for the device is **dimmer_switch.h**
- The clusters supported by the device are listed in Section 3.5.1
- The device structure, `tsZLO_DimmerSwitchDevice`, is listed in Section 3.5.2
- The endpoint registration function for the device, **eZLO_RegisterDimmerSwitchEndPoint()**, is detailed in Section 3.5.3

3.5.1 Supported Clusters

The clusters used by the Dimmer Switch device are listed in the table below.

Server (Input) Side	Client (Output) Side
Mandatory	
Basic	On/Off
Identify	Identify
	Level Control
Optional	
On/Off Switch Configuration	OTA Upgrade
	Scenes
	Groups

Table 6: Clusters for Dimmer Switch

The mandatory attributes within each cluster for this device type are indicated in the *ZigBee Lighting and Occupancy Device Specification (15-0014-01)*.

3.5.2 Device Structure

The following `tsZLO_DimmerSwitchDevice` structure is the shared structure for a Dimmer Switch device:

```
typedef struct
{
    tsZCL_EndPointDefinition sEndPoint;

    /* Cluster instances */
    tsZLO_DimmerSwitchDeviceClusterInstances sClusterInstance;

    /* Mandatory server clusters */
    #if (defined CLD_BASIC) && (defined BASIC_SERVER)
        /* Basic Cluster - Server */
        tsCLD_Basic sBasicServerCluster;
    #endif

    #if (defined CLD_IDENTIFY) && (defined IDENTIFY_SERVER)
        /* Identify Cluster - Server */
        tsCLD_Identify sIdentifyServerCluster;
        tsCLD_IdentifyCustomDataStructure
                            sIdentifyServerCustomDataStructure;
    #endif

        /* Optional server clusters */
    #if (defined CLD_OOSC) && (defined OOSC_SERVER)
        /* On/Off Switch Configuration Cluster - Server */
        tsCLD_OOSC sOOSCServerCluster;
    #endif

    /* Mandatory client clusters */
    #if (defined CLD_IDENTIFY) && (defined IDENTIFY_CLIENT)
        /* Identify Cluster - Client */
        tsCLD_Identify sIdentifyClientCluster;
        tsCLD_IdentifyCustomDataStructure
                           sIdentifyClientCustomDataStructure;
    #endif

    #if (defined CLD_BASIC) && (defined BASIC_CLIENT)
        /* Basic Cluster - Client */
        tsCLD_Basic sBasicClientCluster;
    #endif

    #if (defined CLD_ONOFF) && (defined ONOFF_CLIENT)
        tsCLD_OnOff sOnOffClientCluster;
    #endif

    #if (defined CLD_LEVEL_CONTROL) && (defined LEVEL_CONTROL_CLIENT)
        /* Level Control Cluster - Client */
```

```
                tsCLD_LevelControl sLevelControlClientCluster;
                tsCLD_LevelControlCustomDataStructure
                                  sLevelControlClientCustomDataStructure;
            #endif

            /* Recommended Optional client clusters */
            #if (defined CLD_SCENES) && (defined SCENES_CLIENT)
                /* Scenes Cluster - Client */
                tsCLD_Scenes sScenesClientCluster;
                tsCLD_ScenesCustomDataStructure sScenesClientCustomDataStructure;
            #endif

            #if (defined CLD_GROUPS) && (defined GROUPS_CLIENT)
                /* Groups Cluster - Client */
                tsCLD_Groups sGroupsClientCluster;
                tsCLD_GroupsCustomDataStructure sGroupsClientCustomDataStructure;
            #endif

            #if (defined CLD_OTA) && (defined OTA_CLIENT)
                tsCLD_AS_Ota sCLD_OTA;
                tsOTA_Common sCLD_OTA_CustomDataStruct;
            #endif
        } tsZLO_DimmerSwitchDevice;
```

3.5.3 Registration Function

The following **eZLO_RegisterDimmerSwitchEndPoint()** function is the endpoint registration function for a Dimmer Switch device.

teZCL_Status eZLO_RegisterDimmerSwitchEndPoint(
uint8 *u8EndPointIdentifier*,
tfpZCL_ZCLCallBackFunction *cbCallBack*,
tsZLO_DimmerSwitchDevice **psDeviceInfo*);

Description

This function is used to register an endpoint which will support a Dimmer Switch device. The function must be called after **eZCL_Initialise()**.

The specified identifier for the endpoint is a number in the range 1 to 240 (endpoint 0 is reserved for ZigBee use). Application endpoints are normally numbered consecutively starting at 1. The specified number must be less than or equal to the value of ZLO_NUMBER_OF_ENDPOINTS defined in the **zcl_options.h** file, which represents the highest endpoint number used for application.

As part of this function call, you must specify a user-defined callback function that will be invoked when an event occurs that is associated with the endpoint. This callback function is defined according to the typedef:

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

You must also provide a pointer to a `tsZLO_DimmerSwitchDevice` structure (see Section 3.5.2) which will be used to store all variables relating to the Dimmer Switch device associated with the endpoint. The `sEndPoint` and `sClusterInstance` fields of this structure are set by this function and must not be directly written to by the application.

The function may be called multiple times if more than one endpoint is being used - for example, if more than one Dimmer Switch device is housed in the same hardware, sharing the same module.

Parameters

u8EndPointIdentifier	Endpoint that is to be associated with the registered structure and callback function
cbCallBack	Pointer to the function that will be used to indicate events to the application for this endpoint
psDeviceInfo	Pointer to the structure that will act as storage for all variables related to the device being registered on this endpoint (see Section 3.5.2). The `sEndPoint` and `sClusterInstance` fields are set by this register function for internal use and must not be written to by the application

Returns

E_ZCL_SUCCESS
E_ZCL_FAIL
E_ZCL_ERR_PARAMETER_NULL
E_ZCL_ERR_PARAMETER_RANGE
E_ZCL_ERR_EP_RANGE
E_ZCL_ERR_CLUSTER_0
E_ZCL_ERR_CALLBACK_NULL
E_ZCL_ERR_CLUSTER_NULL
E_ZCL_ERR_SECURITY_RANGE
E_ZCL_ERR_CLUSTER_ID_RANGE
E_ZCL_ERR_MANUFACTURER_SPECIFIC
E_ZCL_ERR_ATTRIBUTE_TYPE_UNSUPPORTED
E_ZCL_ERR_ATTRIBUTE_ID_ORDER
E_ZCL_ERR_ATTRIBUTES_ACCESS

The above codes are described in the *ZCL User Guide (JN-UG-3132)*.

3.6 Colour Dimmer Switch

The Colour Dimmer Switch device is used to control the hue, saturation and luminance of a multi-colour light, and to switch the light device on and off.

- The Device ID is 0x0105
- The header file for the device is **colour_dimmer_switch.h**
- The clusters supported by the device are listed in Section 3.6.1
- The device structure, `tsZLO_ColourDimmerSwitchDevice`, is listed in Section 3.6.2
- The endpoint registration function for the device, **eZLO_RegisterColourDimmerSwitchEndPoint()**, is detailed in Section 3.6.3

3.6.1 Supported Clusters

The clusters used by the Colour Dimmer Switch device are listed in the table below.

Server (Input) Side	Client (Output) Side
Mandatory	
Basic	On/Off
Identify	Level Control
	Colour Control
	Identify
Optional	
On/Off Switch Configuration	OTA Upgrade
	Scenes
	Groups

Table 7: Clusters for Colour Dimmer Switch

The mandatory attributes within each cluster for this device type are indicated in the *ZigBee Lighting and Occupancy Device Specification (15-0014-01)*.

3.6.2 Device Structure

The following `tsZLO_ColourDimmerSwitchDevice` structure is the shared structure for a Colour Dimmer Switch device:

```
typedef struct
{
    tsZCL_EndPointDefinition sEndPoint;

    /* Cluster instances */
    tsZLO_ColourDimmerSwitchDeviceClusterInstances sClusterInstance;

    /* Mandatory server clusters */
    #if (defined CLD_BASIC) && (defined BASIC_SERVER)
        /* Basic Cluster - Server */
        tsCLD_Basic sBasicServerCluster;
    #endif

    #if (defined CLD_IDENTIFY) && (defined IDENTIFY_SERVER)
        /* Identify Cluster - Server */
        tsCLD_Identify sIdentifyServerCluster;
        tsCLD_IdentifyCustomDataStructure
                       sIdentifyServerCustomDataStructure;
    #endif

    /* Optional server clusters */
    #if (defined CLD_OOSC) && (defined OOSC_SERVER)
        /* On/Off Switch Configuration Cluster - Server */
        tsCLD_OOSC sOOSCServerCluster;
    #endif

    /* Mandatory client clusters */
    #if (defined CLD_IDENTIFY) && (defined IDENTIFY_CLIENT)
        /* Identify Cluster - Client */
        tsCLD_Identify sIdentifyClientCluster;
        tsCLD_IdentifyCustomDataStructure
                       sIdentifyClientCustomDataStructure;
    #endif

    #if (defined CLD_BASIC) && (defined BASIC_CLIENT)
        /* Basic Cluster - Server */
        tsCLD_Basic sBasicClientCluster;
    #endif

    #if (defined CLD_ONOFF) && (defined ONOFF_CLIENT)
        tsCLD_OnOff sOnOffClientCluster;
    #endif

    #if (defined CLD_LEVEL_CONTROL) && (defined LEVEL_CONTROL_CLIENT)
        /* Level Control Cluster - Client */
        tsCLD_LevelControl sLevelControlClientCluster;
        tsCLD_LevelControlCustomDataStructure
                           sLevelControlClientCustomDataStructure;
    #endif

    #if (defined CLD_COLOUR_CONTROL) && (defined COLOUR_CONTROL_CLIENT)
        /* Colour Control Cluster - Client */
        tsCLD_ColourControl sColourControlClientCluster;
        tsCLD_ColourControlCustomDataStructure
                            sColourControlClientCustomDataStructure;
    #endif

    /*Recommended Optional client clusters */
    #if (defined CLD_SCENES) && (defined SCENES_CLIENT)
        /* Scenes Cluster - Client */
        tsCLD_Scenes sScenesClientCluster;
        tsCLD_ScenesCustomDataStructure sScenesClientCustomDataStructure;
    #endif

    #if (defined CLD_GROUPS) && (defined GROUPS_CLIENT)
        /* Groups Cluster - Client */
        tsCLD_Groups sGroupsClientCluster;
        tsCLD_GroupsCustomDataStructure sGroupsClientCustomDataStructure;
    #endif

    #if (defined CLD_OTA) && (defined OTA_CLIENT)
        tsCLD_AS_Ota sCLD_OTA;
        tsOTA_Common sCLD_OTA_CustomDataStruct;
    #endif

} tsZLO_ColourDimmerSwitchDevice;
```

3.6.3 Registration Function

The following **eZLO_RegisterColourDimmerSwitchEndPoint()** function is the endpoint registration function for a Colour Dimmer Switch device.

> **teZCL_Status eZLO_RegisterColourDimmerSwitchEndPoint(**
> **uint8** *u8EndPointIdentifier*,
> **tfpZCL_ZCLCallBackFunction** *cbCallBack*,
> **tsZLO_DimmerSwitchDevice** **psDeviceInfo*);

Description

This function is used to register an endpoint which will support a Colour Dimmer Switch device. The function must be called after **eZCL_Initialise()**.

The specified identifier for the endpoint is a number in the range 1 to 240 (endpoint 0 is reserved for ZigBee use). Application endpoints are normally numbered consecutively starting at 1. The specified number must be less than or equal to the value of ZLO_NUMBER_OF_ENDPOINTS defined in the **zcl_options.h** file, which represents the highest endpoint number used for applications.

As part of this function call, you must specify a user-defined callback function that will be invoked when an event occurs that is associated with the endpoint. This callback function is defined according to the typedef:

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

You must also provide a pointer to a `tsZLO_ColourDimmerSwitchDevice` structure (see Section 3.6.2) which will be used to store all variables relating to the Colour Dimmer Switch device associated with the endpoint. The `sEndPoint` and `sClusterInstance` fields of this structure are set by this function and must not be directly written to by the application.

The function may be called multiple times if more than one endpoint is being used - for example, if more than one Colour Dimmer Switch device is housed in the same hardware, sharing the same module.

Parameters

u8EndPointIdentifier	Endpoint that is to be associated with the registered structure and callback function
cbCallBack	Pointer to the function that will be used to indicate events to the application for this endpoint
psDeviceInfo	Pointer to the structure that will act as storage for all variables related to the device being registered on this endpoint (see Section 3.6.2). The `sEndPoint` and `sClusterInstance` fields are set by this register function for internal use and must not be written to by the application

Returns

E_ZCL_SUCCESS
E_ZCL_FAIL
E_ZCL_ERR_PARAMETER_NULL
E_ZCL_ERR_PARAMETER_RANGE
E_ZCL_ERR_EP_RANGE
E_ZCL_ERR_CLUSTER_0
E_ZCL_ERR_CALLBACK_NULL
E_ZCL_ERR_CLUSTER_NULL
E_ZCL_ERR_SECURITY_RANGE
E_ZCL_ERR_CLUSTER_ID_RANGE
E_ZCL_ERR_MANUFACTURER_SPECIFIC
E_ZCL_ERR_ATTRIBUTE_TYPE_UNSUPPORTED
E_ZCL_ERR_ATTRIBUTE_ID_ORDER
E_ZCL_ERR_ATTRIBUTES_ACCESS

The above codes are described in the *ZCL User Guide (JN-UG-3132)*.

3.7 Light Sensor

The Light Sensor device reports the illumination level in an area.

- The Device ID is 0x0106
- The header file for the device is **light_sensor.h**
- The clusters supported by the device are listed in Section 3.7.1
- The device structure, `tsZLO_LightSensorDevice`, is listed in Section 3.7.2
- The endpoint registration function for the device, **eZLO_RegisterLightSensorEndPoint()**, is detailed in Section 3.7.3

3.7.1 Supported Clusters

The clusters used by the Light Sensor device are listed in the table below.

Server (Input) Side	Client (Output) Side
Mandatory	
Basic	Identify
Identify	
Illuminance Measurement	
Optional	
	OTA Upgrade
	Groups

Table 8: Clusters for Light Sensor

The mandatory attributes within each cluster for this device type are indicated in the *ZigBee Lighting and Occupancy Device Specification (15-0014-01)*.

3.7.2 Device Structure

The following `tsZLO_LightSensorDevice` structure is the shared structure for a Light Sensor device:

```
typedef struct
{
    tsZCL_EndPointDefinition sEndPoint;

    /* Cluster instances */
    tsZLO_LightSensorDeviceClusterInstances sClusterInstance;

    /* Mandatory server clusters */
    #if (defined CLD_BASIC) && (defined BASIC_SERVER)
        /* Basic Cluster - Server */
        tsCLD_Basic sBasicServerCluster;
    #endif

    #if (defined CLD_IDENTIFY) && (defined IDENTIFY_SERVER)
        /* Identify Cluster - Server */
        tsCLD_Identify sIdentifyServerCluster;
        tsCLD_IdentifyCustomDataStructure
                      sIdentifyServerCustomDataStructure;
    #endif

    #if (defined CLD_ILLUMINANCE_MEASUREMENT) && (defined
ILLUMINANCE_MEASUREMENT_SERVER)
        /* Illuminance Measurement Cluster - Server */
        tsCLD_IlluminanceMeasurement sIlluminanceMeasurementServerCluster;
    #endif

    /* Optional server clusters */
    #if (defined CLD_POLL_CONTROL) && (defined POLL_CONTROL_SERVER)
        tsCLD_PollControl sPollControlServerCluster;
        tsCLD_PollControlCustomDataStructure
                          sPollControlServerCustomDataStructure;
    #endif

        /* Mandatory server clusters */
    #if (defined CLD_IDENTIFY) && (defined IDENTIFY_CLIENT)
        /* Identify Cluster - Client */
        tsCLD_Identify sIdentifyClientCluster;
        tsCLD_IdentifyCustomDataStructure
                       sIdentifyClientCustomDataStructure;
    #endif

    /* Recommended Optional client clusters */
    #if (defined CLD_GROUPS) && (defined GROUPS_CLIENT)
        /* Groups Cluster - Client */
        tsCLD_Groups sGroupsClientCluster;
        tsCLD_GroupsCustomDataStructure sGroupsClientCustomDataStructure;
    #endif

    #if (defined CLD_OTA) && (defined OTA_CLIENT)
        tsCLD_AS_Ota sCLD_OTA;
        tsOTA_Common sCLD_OTA_CustomDataStruct;
    #endif

} tsZLO_LightSensorDevice;
```

3.7.3 Registration Function

The following **eZLO_RegisterLightSensorEndPoint()** function is the endpoint registration function for a Light Sensor device.

> **teZCL_Status eZLO_RegisterLightSensorEndPoint(**
>   **uint8** *u8EndPointIdentifier*,
>   **tfpZCL_ZCLCallBackFunction** *cbCallBack*,
>   **tsZLO_LightSensorDevice** **psDeviceInfo***);**

Description

This function is used to register an endpoint which will support a Light Sensor device. The function must be called after **eZCL_Initialise()**.

The specified identifier for the endpoint is a number in the range 1 to 240 (endpoint 0 is reserved for ZigBee use). Application endpoints are normally numbered consecutively starting at 1. The specified number must be less than or equal to the value of ZLO_NUMBER_OF_ENDPOINTS defined in the **zcl_options.h** file, which represents the highest endpoint number used for applications.

As part of this function call, you must specify a user-defined callback function that will be invoked when an event occurs that is associated with the endpoint. This callback function is defined according to the typedef:

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

You must also provide a pointer to a `tsZLO_LightSensorDevice` structure (see Section 3.7.2) which will be used to store all variables relating to the Light Sensor device associated with the endpoint. The `sEndPoint` and `sClusterInstance` fields of this structure are set by this function and must not be directly written to by the application.

The function may be called multiple times if more than one endpoint is being used - for example, if more than one Light Sensor device is housed in the same hardware, sharing the same module.

Parameters

u8EndPointIdentifier	Endpoint that is to be associated with the registered structure and callback function
cbCallBack	Pointer to the function that will be used to indicate events to the application for this endpoint
psDeviceInfo	Pointer to the structure that will act as storage for all variables related to the device being registered on this endpoint (see Section 3.7.2). The `sEndPoint` and `sClusterInstance` fields are set by this register function for internal use and must not be written to by the application

Returns

E_ZCL_SUCCESS
E_ZCL_FAIL
E_ZCL_ERR_PARAMETER_NULL
E_ZCL_ERR_PARAMETER_RANGE
E_ZCL_ERR_EP_RANGE
E_ZCL_ERR_CLUSTER_0
E_ZCL_ERR_CALLBACK_NULL
E_ZCL_ERR_CLUSTER_NULL
E_ZCL_ERR_SECURITY_RANGE
E_ZCL_ERR_CLUSTER_ID_RANGE
E_ZCL_ERR_MANUFACTURER_SPECIFIC
E_ZCL_ERR_ATTRIBUTE_TYPE_UNSUPPORTED
E_ZCL_ERR_ATTRIBUTE_ID_ORDER
E_ZCL_ERR_ATTRIBUTES_ACCESS

The above codes are described in the *ZCL User Guide (JN-UG-3132)*.

3.8 Occupancy Sensor

The Occupancy Sensor device reports the presence (or not) of occupants in an area.

- The Device ID is 0x0107
- The header file for the device is **occupancy_sensor.h**
- The clusters supported by the device are listed in Section 3.8.1
- The device structure, `tsZLO_OccupancySensorDevice`, is listed in Section 3.7.2
- The endpoint registration function for the device, **eZLO_RegisterOccupancySensorEndPoint()**, is detailed in Section 3.7.3

3.8.1 Supported Clusters

The clusters used by the Occupancy Sensor device are listed in the table below.

Server (Input) Side	Client (Output) Side
Mandatory	
Basic	Identify
Identify	
Occupancy Sensing	
Optional	
	OTA Upgrade
	Groups

Table 9: Clusters for Occupancy Sensor

The mandatory attributes within each cluster for this device type are indicated in the *ZigBee Lighting and Occupancy Device Specification (15-0014-01)*.

3.8.2 Device Structure

The following `tsZLO_OccupancySensorDevice` structure is the shared structure for an Occupancy Sensor device:

```
typedef struct
{
    tsZCL_EndPointDefinition sEndPoint;

    /* Cluster instances */
    tsZLO_OccupancySensorDeviceClusterInstances sClusterInstance;

    /* Mandatory server clusters */
#if (defined CLD_BASIC) && (defined BASIC_SERVER)
```

```
    /* Basic Cluster - Server */
    tsCLD_Basic sBasicServerCluster;
#endif

#if (defined CLD_IDENTIFY) && (defined IDENTIFY_SERVER)
    /* Identify Cluster - Server */
    tsCLD_Identify sIdentifyServerCluster;
    tsCLD_IdentifyCustomDataStructure sIdentifyServerCustomDataStructure;
#endif

#if (defined CLD_OCCUPANCY_SENSING) && (defined OCCUPANCY_SENSING_SERVER)
    /* Occupancy Sensing Cluster - Server */
    tsCLD_OccupancySensing sOccupancySensingServerCluster;
#endif

    /* Optional server clusters */
#if (defined CLD_POLL_CONTROL) && (defined POLL_CONTROL_SERVER)
    tsCLD_PollControl sPollControlServerCluster;
    tsCLD_PollControlCustomDataStructure
                      sPollControlServerCustomDataStructure;
#endif

    /* Mandatory client clusters */
#if (defined CLD_IDENTIFY) && (defined IDENTIFY_CLIENT)
    /* Identify Cluster - Client */
    tsCLD_Identify sIdentifyClientCluster;
    tsCLD_IdentifyCustomDataStructure sIdentifyClientCustomDataStructure;
#endif

    /* Recommended Optional client clusters */
#if (defined CLD_POLL_CONTROL) && (defined POLL_CONTROL_CLIENT)
    tsCLD_PollControl sPollControlClientCluster;
    tsCLD_PollControlCustomDataStructure
                      sPollControlClientCustomDataStructure;
#endif

#if (defined CLD_GROUPS) && (defined GROUPS_CLIENT)
    /* Groups Cluster - Client */
    tsCLD_Groups sGroupsClientCluster;
    tsCLD_GroupsCustomDataStructure sGroupsClientCustomDataStructure;
#endif

#if (defined CLD_OTA) && (defined OTA_CLIENT)
    tsCLD_AS_Ota sCLD_OTA;
    tsOTA_Common sCLD_OTA_CustomDataStruct;
#endif

} tsZLO_OccupancySensorDevice;;
```

3.8.3 Registration Function

The following **eZLO_RegisterOccupancySensorEndPoint()** function is the endpoint registration function for an Occupancy Sensor device.

teZCL_Status eZLO_RegisterOccupancySensorEndPoint(
    uint8 *u8EndPointIdentifier*,
    tfpZCL_ZCLCallBackFunction *cbCallBack*,
    tsZLO_OccupancySensorDevice **psDeviceInfo*);

Description

This function is used to register an endpoint which will support an Occupancy Sensor device. The function must be called after **eZCL_Initialise()**.

The specified identifier for the endpoint is a number in the range 1 to 240 (endpoint 0 is reserved for ZigBee use). Application endpoints are normally numbered consecutively starting at 1. The specified number must be less than or equal to the value of ZLO_NUMBER_OF_ENDPOINTS defined in the **zcl_options.h** file, which represents the highest endpoint number used for applications.

As part of this function call, you must specify a user-defined callback function that will be invoked when an event occurs that is associated with the endpoint. This callback function is defined according to the typedef:

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

You must also provide a pointer to a `tsZLO_OccupancySensorDevice` structure (see Section 3.8.2) which will be used to store all variables relating to the Light Sensor device associated with the endpoint. The `sEndPoint` and `sClusterInstance` fields of this structure are set by this function and must not be directly written to by the application.

The function may be called multiple times if more than one endpoint is being used - for example, if more than one Occupancy Sensor device is housed in the same hardware, sharing the same module.

Parameters

u8EndPointIdentifier	Endpoint that is to be associated with the registered structure and callback function
cbCallBack	Pointer to the function that will be used to indicate events to the application for this endpoint
psDeviceInfo	Pointer to the structure that will act as storage for all variables related to the device being registered on this endpoint (see Section 3.8.2). The `sEndPoint` and `sClusterInstance` fields are set by this register function for internal use and must not be written to by the application

Returns

E_ZCL_SUCCESS
E_ZCL_FAIL
E_ZCL_ERR_PARAMETER_NULL
E_ZCL_ERR_PARAMETER_RANGE
E_ZCL_ERR_EP_RANGE
E_ZCL_ERR_CLUSTER_0
E_ZCL_ERR_CALLBACK_NULL
E_ZCL_ERR_CLUSTER_NULL
E_ZCL_ERR_SECURITY_RANGE
E_ZCL_ERR_CLUSTER_ID_RANGE
E_ZCL_ERR_MANUFACTURER_SPECIFIC
E_ZCL_ERR_ATTRIBUTE_TYPE_UNSUPPORTED
E_ZCL_ERR_ATTRIBUTE_ID_ORDER
E_ZCL_ERR_ATTRIBUTES_ACCESS

The above codes are described in the *ZCL User Guide (JN-UG-3132)*.

3.9 On/Off Ballast

The On/Off Ballast is a lighting device that can be switched on/off from a controller device, such as an On/Off Light Switch or an Occupancy Sensor.

- The Device ID is 0x0108
- The header file for the device is **on_off_ballast.h**
- The clusters supported by the device are listed in Section 3.9.1
- The device structure, `tsZLO_OnOffBallastDevice`, is listed in Section 3.9.2
- The endpoint registration function for the device, **eZLO_RegisterOnOffBallastEndPoint()**, is detailed in Section 3.9.3

3.9.1 Supported Clusters

The clusters used by the On/Off Ballast device are listed in the table below.

Server (Input) Side	Client (Output) Side
Mandatory	
Basic	
Power Configuration	
Device Temperature Configuration	
Identify	
Groups	
Scenes	
On/Off	
Ballast Configuration	
Optional	
Level Control	OTA Upgrade
Illuminance Level Sensing	Illuminance Measurement
Touchlink Commissioning	Illuminance Level Sensing
	Occupancy Sensing

Table 10: Clusters for On/Off Ballast

The mandatory attributes within each cluster for this device type are indicated in the *ZigBee Lighting and Occupancy Device Specification (15-0014-01)*.

3.9.2 Device Structure

The following `tsZLO_OnOffBallastDevice` structure is the shared structure for an On/Off Ballast device:

```
typedef struct
{
    tsZCL_EndPointDefinition sEndPoint;

    /* Cluster instances */
    tsZLO_OnOffBallastDeviceClusterInstances sClusterInstance;

    /* Mandatory server clusters */
    #if (defined CLD_BASIC) && (defined BASIC_SERVER)
        /* Basic Cluster - Server */
        tsCLD_Basic sBasicServerCluster;
    #endif

    #if (defined CLD_IDENTIFY) && (defined IDENTIFY_SERVER)
        /* Identify Cluster - Server */
        tsCLD_Identify sIdentifyServerCluster;
        tsCLD_IdentifyCustomDataStructure
                            sIdentifyServerCustomDataStructure;
    #endif

    #if (defined CLD_ONOFF) && (defined ONOFF_SERVER)
        /* On/Off Cluster - Server */
        tsCLD_OnOff sOnOffServerCluster;
        tsCLD_OnOffCustomDataStructure sOnOffServerCustomDataStructure;
    #endif

    #if (defined CLD_SCENES) && (defined SCENES_SERVER)
        /* Scenes Cluster - Server */
        tsCLD_Scenes sScenesServerCluster;
        tsCLD_ScenesCustomDataStructure sScenesServerCustomDataStructure;
    #endif

    #if (defined CLD_GROUPS) && (defined GROUPS_SERVER)
        /* Groups Cluster - Server */
        tsCLD_Groups sGroupsServerCluster;
        tsCLD_GroupsCustomDataStructure sGroupsServerCustomDataStructure;
    #endif

    /* Optional server clusters */
    #if (defined CLD_POWER_CONFIGURATION) && (defined
POWER_CONFIGURATION_SERVER)
        /* Power Configuration Cluster - Server */
        tsCLD_PowerConfiguration sPowerConfigServerCluster;
    #endif
    #if (defined CLD_DEVICE_TEMPERATURE_CONFIGURATION) && (defined
DEVICE_TEMPERATURE_CONFIGURATION_SERVER)
        /* Device Temperature Configuration Cluster - Server */
        tsCLD_DeviceTemperatureConfiguration
                          sDeviceTemperatureConfigurationServerCluster;
    #endif

    #if (defined CLD_BALLAST_CONFIGURATION) && (defined
BALLAST_CONFIGURATION_SERVER)
        tsCLD_BallastConfiguration sBallastConfigurationServerCluster;
    #endif

    /* Recommended Optional server clusters */
    #if (defined CLD_LEVEL_CONTROL) && (defined LEVEL_CONTROL_SERVER)
        /* LevelControl Cluster - Server */
        tsCLD_LevelControl sLevelControlServerCluster;
        tsCLD_LevelControlCustomDataStructure
                               sLevelControlServerCustomDataStructure;
    #endif

    #if (defined CLD_ILLUMINANCE_LEVEL_SENSING) && (defined
ILLUMINANCE_LEVEL_SENSING_SERVER)
        tsCLD_IlluminanceLevelSensing
sIlluminanceLevelSensingServerCluster;
    #endif

    #if (defined CLD_ZLL_COMMISSION) && (defined ZLL_COMMISSION_SERVER)
        tsCLD_ZllCommission sZllCommissionServerCluster;
        tsCLD_ZllCommissionCustomDataStructure
                          sZllCommissionServerCustomDataStructure;
    #endif

    /*Recommended Optional client clusters */
    #if (defined CLD_OTA) && (defined OTA_CLIENT)
        /* OTA cluster - Client */
        tsCLD_AS_Ota sCLD_OTA;
        tsOTA_Common sCLD_OTA_CustomDataStruct;
    #endif

    #if (defined CLD_ILLUMINANCE_MEASUREMENT) && (defined
ILLUMINANCE_MEASUREMENT_CLIENT)
        /* Illuminance Measurement Cluster - Client */
        tsCLD_IlluminanceMeasurement sIlluminanceMeasurementClientCluster;
    #endif

    #if (defined CLD_ILLUMINANCE_LEVEL_SENSING) && (defined
ILLUMINANCE_LEVEL_SENSING_CLIENT)
        tsCLD_IlluminanceLevelSensing
                          sIlluminanceLevelSensingClientCluster;
    #endif
    #if (defined CLD_OCCUPANCY_SENSING) && (defined
OCCUPANCY_SENSING_CLIENT)
        /* Occupancy Sensing Cluster - Client */
        tsCLD_OccupancySensing sOccupancySensingClientCluster;
    #endif

} tsZLO_OnOffBallastDevice;
```

3.9.3 Registration Function

The following **eZLO_RegisterOnOffBallastEndPoint()** function is the endpoint registration function for an On/Off Ballast device.

teZCL_Status eZLO_RegisterOnOffBallastEndPoint(
uint8 *u8EndPointIdentifier*,
tfpZCL_ZCLCallBackFunction *cbCallBack*,
tsZLO_OnOffBallastDevice **psDeviceInfo*);

Description

This function is used to register an endpoint which will support an On/Off Ballast device. The function must be called after **eZCL_Initialise()**.

The specified identifier for the endpoint is a number in the range 1 to 240 (endpoint 0 is reserved for ZigBee use). Application endpoints are normally numbered consecutively starting at 1. The specified number must be less than or equal to the value of ZLO_NUMBER_OF_ENDPOINTS defined in the **zcl_options.h** file, which represents the highest endpoint number used for applications.

As part of this function call, you must specify a user-defined callback function that will be invoked when an event occurs that is associated with the endpoint. This callback function is defined according to the typedef:

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

You must also provide a pointer to a `tsZLO_OnOffBallastDevice` structure (see Section 3.9.2) which will be used to store all variables relating to the On/Off Ballast device associated with the endpoint. The `sEndPoint` and `sClusterInstance` fields of this structure are set by this function and must not be directly written to by the application.

The function may be called multiple times if more than one endpoint is being used - for example, if more than one On/Off Ballast device is housed in the same hardware, sharing the same module.

Parameters

u8EndPointIdentifier	Endpoint that is to be associated with the registered structure and callback function
cbCallBack	Pointer to the function that will be used to indicate events to the application for this endpoint
psDeviceInfo	Pointer to the structure that will act as storage for all variables related to the device being registered on this endpoint (see Section 3.9.2). The `sEndPoint` and `sClusterInstance` fields are set by this register function for internal use and must not be written to by the application

Returns

E_ZCL_SUCCESS
E_ZCL_FAIL
E_ZCL_ERR_PARAMETER_NULL
E_ZCL_ERR_PARAMETER_RANGE
E_ZCL_ERR_EP_RANGE
E_ZCL_ERR_CLUSTER_0
E_ZCL_ERR_CALLBACK_NULL
E_ZCL_ERR_CLUSTER_NULL
E_ZCL_ERR_SECURITY_RANGE
E_ZCL_ERR_CLUSTER_ID_RANGE
E_ZCL_ERR_MANUFACTURER_SPECIFIC
E_ZCL_ERR_ATTRIBUTE_TYPE_UNSUPPORTED
E_ZCL_ERR_ATTRIBUTE_ID_ORDER
E_ZCL_ERR_ATTRIBUTES_ACCESS

The above codes are described in the *ZCL User Guide (JN-UG-3132)*.

3.10 Dimmable Ballast

The Dimmable Ballast is a lighting device that can be switched on/off or have its level adjusted from a controller device, such as a Dimmer Switch, or simply be switched on/off from an Occupancy Sensor.

- The Device ID is 0x0109
- The header file for the device is **dimmable_ballast.h**
- The clusters supported by the device are listed in Section 3.10.1
- The device structure, `tsZLO_DimmableBallastDevice`, is listed in Section 3.10.2
- The endpoint registration function for the device, **eZLO_RegisterDimmableBallastEndPoint**, is detailed in Section 3.10.3

3.10.1 Supported Clusters

The clusters used by the Dimmable Ballast device are listed in the table below.

Server (Input) Side	**Client (Output) Side**
Mandatory	
Basic	
Power Configuration	
Device Temperature Configuration	
Identify	
Groups	
Scenes	
On/Off	
Level Control	
Ballast Configuration	
Optional	
Illuminance Level Sensing	OTA Upgrade
Touchlink Commissioning	Illuminance Measurement
	Illuminance Level Sensing
	Occupancy Sensing

Table 11: Clusters for On/Off Ballast

The mandatory attributes within each cluster for this device type are indicated in the *ZigBee Lighting and Occupancy Device Specification (15-0014-01)*.

3.10.2 Device Structure

The following `tsZLO_DimmableBallastDevice` structure is the shared structure for a Dimmable Ballast device:

```
typedef struct
{
    tsZCL_EndPointDefinition sEndPoint;

    /* Cluster instances */
    tsZLO_DimmableBallastDeviceClusterInstances sClusterInstance;

    /* Mandatory server clusters */
    #if (defined CLD_BASIC) && (defined BASIC_SERVER)
        /* Basic Cluster - Server */
        tsCLD_Basic sBasicServerCluster;
    #endif

    #if (defined CLD_IDENTIFY) && (defined IDENTIFY_SERVER)
        /* Identify Cluster - Server */
        tsCLD_Identify sIdentifyServerCluster;
        tsCLD_IdentifyCustomDataStructure
                       sIdentifyServerCustomDataStructure;
    #endif

    #if (defined CLD_ONOFF) && (defined ONOFF_SERVER)
        /* On/Off Cluster - Server */
        tsCLD_OnOff sOnOffServerCluster;
        tsCLD_OnOffCustomDataStructure sOnOffServerCustomDataStructure;
    #endif

    #if (defined CLD_LEVEL_CONTROL) && (defined LEVEL_CONTROL_SERVER)
        /* LevelControl Cluster - Server */
        tsCLD_LevelControl sLevelControlServerCluster;
        tsCLD_LevelControlCustomDataStructure
sLevelControlServerCustomDataStructure;
    #endif

    #if (defined CLD_SCENES) && (defined SCENES_SERVER)
        /* Scenes Cluster - Server */
        tsCLD_Scenes sScenesServerCluster;
        tsCLD_ScenesCustomDataStructure sScenesServerCustomDataStructure;
    #endif

    #if (defined CLD_GROUPS) && (defined GROUPS_SERVER)
        /* Groups Cluster - Server */
        tsCLD_Groups sGroupsServerCluster;
        tsCLD_GroupsCustomDataStructure sGroupsServerCustomDataStructure;
    #endif
    #if (defined CLD_POWER_CONFIGURATION) && (defined
POWER_CONFIGURATION_SERVER)
        /* Power Configuration Cluster - Server */
        tsCLD_PowerConfiguration sPowerConfigServerCluster;
    #endif

    #if (defined CLD_DEVICE_TEMPERATURE_CONFIGURATION) && (defined
DEVICE_TEMPERATURE_CONFIGURATION_SERVER)
        /* Device Temperature Configuration Cluster - Server */
        tsCLD_DeviceTemperatureConfiguration
                          sDeviceTemperatureConfigurationServerCluster;
    #endif

    #if (defined CLD_BALLAST_CONFIGURATION) && (defined
BALLAST_CONFIGURATION_SERVER)
        tsCLD_BallastConfiguration sBallastConfigurationServerCluster;
    #endif

    /* Recommended Optional server clusters */
    #if (defined CLD_ILLUMINANCE_LEVEL_SENSING) && (defined
ILLUMINANCE_LEVEL_SENSING_SERVER)
        tsCLD_IlluminanceLevelSensing
                                   sIlluminanceLevelSensingServerCluster;
    #endif

    #if (defined CLD_ZLL_COMMISSION) && (defined ZLL_COMMISSION_SERVER)
        tsCLD_ZllCommission sZllCommissionServerCluster;
        tsCLD_ZllCommissionCustomDataStructure
                            sZllCommissionServerCustomDataStructure;
    #endif

    /*Recommended Optional client clusters */
    #if (defined CLD_OTA) && (defined OTA_CLIENT)
        /* OTA cluster - Client */
        tsCLD_AS_Ota sCLD_OTA;
        tsOTA_Common sCLD_OTA_CustomDataStruct;
    #endif

    #if (defined CLD_ILLUMINANCE_MEASUREMENT) && (defined
ILLUMINANCE_MEASUREMENT_CLIENT)
        /* Illuminance Measurement Cluster - Client */
        tsCLD_IlluminanceMeasurement sIlluminanceMeasurementClientCluster;
    #endif

    #if (defined CLD_ILLUMINANCE_LEVEL_SENSING) && (defined
ILLUMINANCE_LEVEL_SENSING_CLIENT)
        tsCLD_IlluminanceLevelSensing
                           sIlluminanceLevelSensingClientCluster;
    #endif
    #if (defined CLD_OCCUPANCY_SENSING) && (defined
OCCUPANCY_SENSING_CLIENT)
        /* Occupancy Sensing Cluster - Client */
        tsCLD_OccupancySensing sOccupancySensingClientCluster;
    #endif

} tsZLO_DimmableBallastDevice;
```

3.10.3 Registration Function

The following **eZLO_RegisterDimmableBallastEndPoint()** function is the endpoint registration function for a Dimmable Ballast device.

teZCL_Status eZLO_RegisterDimmableBallastEndPoint(
uint8 *u8EndPointIdentifier*,
tfpZCL_ZCLCallBackFunction *cbCallBack*,
tsZLO_DimmableBallastDevice **psDeviceInfo*);

Description

This function is used to register an endpoint which will support a Dimmable Ballast device. The function must be called after **eZCL_Initialise()**.

The specified identifier for the endpoint is a number in the range 1 to 240 (endpoint 0 is reserved for ZigBee use). Application endpoints are normally numbered consecutively starting at 1. The specified number must be less than or equal to the value of ZLO_NUMBER_OF_ENDPOINTS defined in the **zcl_options.h** file, which represents the highest endpoint number used for applications.

As part of this function call, you must specify a user-defined callback function that will be invoked when an event occurs that is associated with the endpoint. This callback function is defined according to the typedef:

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

You must also provide a pointer to a `tsZLO_DimmableBallastDevice` structure (see Section 3.10.2) which will be used to store all variables relating to the Dimmable Ballast device associated with the endpoint. The `sEndPoint` and `sClusterInstance` fields of this structure are set by this function and must not be directly written to by the application.

The function may be called multiple times if more than one endpoint is being used - for example, if more than one Dimmable Ballast device is housed in the same hardware, sharing the same module.

Parameters

u8EndPointIdentifier	Endpoint that is to be associated with the registered structure and callback function
cbCallBack	Pointer to the function that will be used to indicate events to the application for this endpoint
psDeviceInfo	Pointer to the structure that will act as storage for all variables related to the device being registered on this endpoint (see Section 3.10.2). The `sEndPoint` and `sClusterInstance` fields are set by this register function for internal use and must not be written to by the application

Returns

E_ZCL_SUCCESS
E_ZCL_FAIL
E_ZCL_ERR_PARAMETER_NULL
E_ZCL_ERR_PARAMETER_RANGE
E_ZCL_ERR_EP_RANGE
E_ZCL_ERR_CLUSTER_0
E_ZCL_ERR_CALLBACK_NULL
E_ZCL_ERR_CLUSTER_NULL
E_ZCL_ERR_SECURITY_RANGE
E_ZCL_ERR_CLUSTER_ID_RANGE
E_ZCL_ERR_MANUFACTURER_SPECIFIC
E_ZCL_ERR_ATTRIBUTE_TYPE_UNSUPPORTED
E_ZCL_ERR_ATTRIBUTE_ID_ORDER
E_ZCL_ERR_ATTRIBUTES_ACCESS

The above codes are described in the *ZCL User Guide (JN-UG-3132)*.

3.11 On/Off Plug-in Unit

The On/Off Plug-in Unit device is typically used in nodes that contain a controllable mains plug or adaptor which includes an on/off switch. It may be controlled from a controller device such as an On/Off Light Switch.

- The Device ID is 0x010A
- The header file for the device is **on_off_plug.h**
- The clusters supported by the device are listed in Section 3.11.1
- The device structure, `tsZLO_OnOffPlugDevice`, is listed in Section 3.11.2
- The endpoint registration function for the device, **eZLO_RegisterOnOffPlugEndPoint()**, is detailed in Section 3.11.3

3.11.1 Supported Clusters

The clusters supported by the On/Off Plug-in Unit device are as follows:

Server (Input) Side	Client (Output) Side
Mandatory	
Basic	
Identify	
Groups	
Scenes	
On/Off	
Optional	
Level Control	OTA Upgrade

Table 12: Clusters for On/Off Plug-in Unit

The mandatory attributes within each cluster for this device type are indicated in the *ZigBee Lighting and Occupancy Device Specification (15-0014-01)*.

3.11.2 Device Structure

The following `tsZLO_OnOffPlugDevice` structure is the shared structure for an On/Off Plug-in Unit device:

```
typedef struct
{
    tsZCL_EndPointDefinition sEndPoint;

    /* Cluster instances */
    tsZLO_OnOffPlugDeviceClusterInstances sClusterInstance;
    #if (defined CLD_BASIC) && (defined BASIC_SERVER)
        /* Basic Cluster - Server */
        tsCLD_Basic sBasicServerCluster;
    #endif

    #if (defined CLD_IDENTIFY) && (defined IDENTIFY_SERVER)
        /* Identify Cluster - Server */
        tsCLD_Identify sIdentifyServerCluster;
        tsCLD_IdentifyCustomDataStructure
                       sIdentifyServerCustomDataStructure;
    #endif

    #if (defined CLD_ONOFF) && (defined ONOFF_SERVER)
        /* On/Off Cluster - Server */
        tsCLD_OnOff sOnOffServerCluster;
        tsCLD_OnOffCustomDataStructure
                    sOnOffServerCustomDataStructure;
    #endif

    #if (defined CLD_GROUPS) && (defined GROUPS_SERVER)
        /* Groups Cluster - Server */
        tsCLD_Groups sGroupsServerCluster;
        tsCLD_GroupsCustomDataStructure
                     sGroupsServerCustomDataStructure;
    #endif

    #if (defined CLD_SCENES) && (defined SCENES_SERVER)
        /* Scenes Cluster - Server */
        tsCLD_Scenes sScenesServerCluster;
        tsCLD_ScenesCustomDataStructure
                     sScenesServerCustomDataStructure;
    #endif

    #if (defined CLD_LEVEL_CONTROL) && (defined LEVEL_CONTROL_SERVER)
        /* LevelControl Cluster - Server */
        tsCLD_LevelControl sLevelControlServerCluster;
        tsCLD_LevelControlCustomDataStructure
                           sLevelControlServerCustomDataStructure;
    #endif

    #if (defined CLD_OTA) && (defined OTA_CLIENT)
        /* OTA cluster - Client */
        tsCLD_AS_Ota sCLD_OTA;
        tsOTA_Common sCLD_OTA_CustomDataStruct;
    #endif
} tsZLO_OnOffPlugDevice;
```

3.11.3 Registration Function

The following **eZLO_RegisterOnOffPlugEndPoint()** function is the endpoint registration function for an On/Off Plug-in Unit device.

teZCL_Status eZLO_RegisterOnOffPlugEndPoint(
uint8 *u8EndPointIdentifier*,
tfpZCL_ZCLCallBackFunction *cbCallBack*,
tsZLO_OnOffPlugDevice **psDeviceInfo*);

Description

This function is used to register an endpoint which will support an On/Off Plug-in Unit device. The function must be called after **eZCL_Initialise()**.

The specified identifier for the endpoint is a number in the range 1 to 240 (endpoint 0 is reserved for ZigBee use). Application endpoints are normally numbered consecutively starting at 1. The specified number must be less than or equal to the value of ZLO_NUMBER_OF_ENDPOINTS defined in the **zcl_options.h** file, which represents the highest endpoint number used for applications.

As part of this function call, you must specify a user-defined callback function that will be invoked when an event occurs that is associated with the endpoint. This callback function is defined according to the typedef:

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

You must also provide a pointer to a `tsZLO_OnOffPlugDevice` structure (see Section 3.11.2) which will be used to store all variables relating to the On/Off Plug-in Unit device associated with the endpoint. The `sEndPoint` and `sClusterInstance` fields of this structure are set by this function and must not be directly written to by the application.

The function may be called multiple times if more than one endpoint is being used - for example, if more than one On/Off Plug-in Unit device is housed in the same hardware, sharing the same module.

Parameters

u8EndPointIdentifier	Endpoint that is to be associated with the registered structure and callback function
cbCallBack	Pointer to the function that will be used to indicate events to the application for this endpoint
psDeviceInfo	Pointer to the structure that will act as storage for all variables related to the device being registered on this endpoint (see Section 3.11.2). The `sEndPoint` and `sClusterInstance` fields are set by this register function for internal use and must not be written to by the application

Returns

E_ZCL_SUCCESS
E_ZCL_FAIL
E_ZCL_ERR_PARAMETER_NULL
E_ZCL_ERR_PARAMETER_RANGE
E_ZCL_ERR_EP_RANGE
E_ZCL_ERR_CLUSTER_0
E_ZCL_ERR_CALLBACK_NULL
E_ZCL_ERR_CLUSTER_NULL
E_ZCL_ERR_SECURITY_RANGE
E_ZCL_ERR_CLUSTER_ID_RANGE
E_ZCL_ERR_MANUFACTURER_SPECIFIC
E_ZCL_ERR_ATTRIBUTE_TYPE_UNSUPPORTED
E_ZCL_ERR_ATTRIBUTE_ID_ORDER
E_ZCL_ERR_ATTRIBUTES_ACCESS

The above codes are described in the *ZCL User Guide (JN-UG-3132)*.

3.12 Dimmable Plug-in Unit

The Dimmable Plug-in Unit device is typically used in nodes that contain a controllable mains plug or adaptor which includes an adjustable output (to a lamp). It may be controlled from a controller device such as a Dimmer Switch or a Non-colour Controller.

- The Device ID is 0x010B
- The header file for the device is **dimmable_plug.h**
- The clusters supported by the device are listed in Section 3.12.1
- The device structure, `tsZLO_DimmablePlugDevice`, is listed in Section 3.12.2
- The endpoint registration function for the device, **eZLO_RegisterDimmablePlugEndPoint()**, is detailed in Section 3.12.3

3.12.1 Supported Clusters

The clusters supported by the Dimmable Plug-in Unit device are as follows:

Server (Input) Side	Client (Output) Side
Mandatory	
Basic	
Identify	
Groups	
Scenes	
On/Off	
Level Control	
Optional	
	OTA Upgrade

Table 13: Clusters for Dimmable Plug-in Unit

The mandatory attributes within each cluster for this device type are indicated in the *ZigBee Lighting and Occupancy Device Specification (15-0014-01)*.

3.12.2 Device Structure

The following `tsZLO_DimmablePlugDevice` structure is the shared structure for a Dimmable Plug-in Unit device:

```
typedef struct
{
    tsZCL_EndPointDefinition sEndPoint;
    /* Cluster instances */
    tsZLO_DimmablePlugDeviceClusterInstances sClusterInstance;

    #if (defined CLD_BASIC) && (defined BASIC_SERVER)
        /* Basic Cluster - Server */
        tsCLD_Basic sBasicServerCluster;
    #endif

    #if (defined CLD_IDENTIFY) && (defined IDENTIFY_SERVER)
        /* Identify Cluster - Server */
        tsCLD_Identify sIdentifyServerCluster;
        tsCLD_IdentifyCustomDataStructure
                       sIdentifyServerCustomDataStructure;
    #endif

    #if (defined CLD_ONOFF) && (defined ONOFF_SERVER)
        /* On/Off Cluster - Server */
        tsCLD_OnOff sOnOffServerCluster;
        tsCLD_OnOffCustomDataStructure sOnOffServerCustomDataStructure;
    #endif

    #if (defined CLD_GROUPS) && (defined GROUPS_SERVER)
        /* Groups Cluster - Server */
        tsCLD_Groups sGroupsServerCluster;
        tsCLD_GroupsCustomDataStructure sGroupsServerCustomDataStructure;
    #endif

    #if (defined CLD_SCENES) && (defined SCENES_SERVER)
        /* Scenes Cluster - Server */
        tsCLD_Scenes sScenesServerCluster;
        tsCLD_ScenesCustomDataStructure sScenesServerCustomDataStructure;
    #endif

    #if (defined CLD_LEVEL_CONTROL) && (defined LEVEL_CONTROL_SERVER)
        /* LevelControl Cluster - Server */
        tsCLD_LevelControl sLevelControlServerCluster;
        tsCLD_LevelControlCustomDataStructure
                           sLevelControlServerCustomDataStructure;
    #endif

    #if (defined CLD_OTA) && (defined OTA_CLIENT)
        /* OTA cluster - Client */
        tsCLD_AS_Ota sCLD_OTA;
        tsOTA_Common sCLD_OTA_CustomDataStruct;
    #endif
} tsZLO_DimmablePlugDevice;
```

3.12.3 Registration Function

The following **eZLO_RegisterDimmablePlugEndPoint()** function is the endpoint registration function for a Dimmable Plug-in Unit device.

teZCL_Status eZLO_RegisterDimmablePlugEndPoint(
uint8 *u8EndPointIdentifier*,
tfpZCL_ZCLCallBackFunction *cbCallBack*,
tsZLO_DimmablePlugDevice **psDeviceInfo*);

Description

This function is used to register an endpoint which will support a Dimmable Plug-in Unit device. The function must be called after **eZCL_Initialise()**.

The specified identifier for the endpoint is a number in the range 1 to 240 (endpoint 0 is reserved for ZigBee use). Application endpoints are normally numbered consecutively starting at 1. The specified number must be less than or equal to the value of ZLO_NUMBER_OF_ENDPOINTS defined in the **zcl_options.h** file, which represents the highest endpoint number used for applications.

As part of this function call, you must specify a user-defined callback function that will be invoked when an event occurs that is associated with the endpoint. This callback function is defined according to the typedef:

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

You must also provide a pointer to a `tsZLO_DimmablePlugDevice` structure (see Section 3.12.2) which will be used to store all variables relating to the Dimmable Plug-in Unit device associated with the endpoint. The `sEndPoint` and `sClusterInstance` fields of this structure are set by this function and must not be directly written to by the application.

The function may be called multiple times if more than one endpoint is being used - for example, if more than one Dimmable Plug-in Unit device is housed in the same hardware, sharing the same module.

Parameters

u8EndPointIdentifier	Endpoint that is to be associated with the registered structure and callback function
cbCallBack	Pointer to the function that will be used to indicate events to the application for this endpoint
psDeviceInfo	Pointer to the structure that will act as storage for all variables related to the device being registered on this endpoint (see Section 3.12.2). The `sEndPoint` and `sClusterInstance` fields are set by this register function for internal use and must not be written to by the application

Returns

E_ZCL_SUCCESS
E_ZCL_FAIL
E_ZCL_ERR_PARAMETER_NULL
E_ZCL_ERR_PARAMETER_RANGE
E_ZCL_ERR_EP_RANGE
E_ZCL_ERR_CLUSTER_0
E_ZCL_ERR_CALLBACK_NULL
E_ZCL_ERR_CLUSTER_NULL
E_ZCL_ERR_SECURITY_RANGE
E_ZCL_ERR_CLUSTER_ID_RANGE
E_ZCL_ERR_MANUFACTURER_SPECIFIC
E_ZCL_ERR_ATTRIBUTE_TYPE_UNSUPPORTED
E_ZCL_ERR_ATTRIBUTE_ID_ORDER
E_ZCL_ERR_ATTRIBUTES_ACCESS

The above codes are described in the *ZCL User Guide (JN-UG-3132)*.

3.13 Colour Temperature Light

The Colour Temperature Light device is typically used in nodes that contain a colour lamp with adjustable colour (and brightness) which operates using colour temperature.

- The Device ID is 0x010C
- The header file for the device is **colour_temperature_light.h**
- The clusters supported by the device are listed in Section 3.13.1
- The device structure, `tsZLO_ColourTemperatureLightDevice`, is listed in Section 3.13.2
- The endpoint registration function for the device, **eZLO_RegisterColourTemperatureLightEndPoint()**, is detailed in Section 3.13.3

3.13.1 Supported Clusters

The clusters supported by the Colour Temperature Light device are as follows:

Server (Input) Side	Client (Output) Side
Mandatory	
Basic	
Identify	
Groups	
Scenes	
On/Off	
Level Control	
Colour Control	
Optional	
Touchlink Commissioning	OTA Upgrade

Table 14: Clusters for Colour Temperature Light

The mandatory attributes within each cluster for this device type are indicated in the *ZigBee Lighting and Occupancy Device Specification (15-0014-01)*.

3.13.2 Device Structure

The following `tsZLO_ColourTemperatureLightDevice` structure is the shared structure for a Colour Temperature Light device:

```
typedef struct
{
    tsZCL_EndPointDefinition sEndPoint;

    /* Cluster instances */
    tsZLO_ColourTemperatureLightDeviceClusterInstances sClusterInstance;

    #if (defined CLD_BASIC) && (defined BASIC_SERVER)
        /* Basic Cluster - Server */
        tsCLD_Basic sBasicServerCluster;
    #endif

    #if (defined CLD_IDENTIFY) && (defined IDENTIFY_SERVER)
        /* Identify Cluster - Server */
        tsCLD_Identify sIdentifyServerCluster;
        tsCLD_IdentifyCustomDataStructure
                       sIdentifyServerCustomDataStructure;
    #endif

    #if (defined CLD_ONOFF) && (defined ONOFF_SERVER)
        /* On/Off Cluster - Server */
        tsCLD_OnOff sOnOffServerCluster;
        tsCLD_OnOffCustomDataStructure sOnOffServerCustomDataStructure;
    #endif

    #if (defined CLD_GROUPS) && (defined GROUPS_SERVER)
        /* Groups Cluster - Server */
        tsCLD_Groups sGroupsServerCluster;
        tsCLD_GroupsCustomDataStructure sGroupsServerCustomDataStructure;
    #endif

    #if (defined CLD_SCENES) && (defined SCENES_SERVER)
        /* Scenes Cluster - Server */
        tsCLD_Scenes sScenesServerCluster;
        tsCLD_ScenesCustomDataStructure sScenesServerCustomDataStructure;
    #endif

    #if (defined CLD_LEVEL_CONTROL) && (defined LEVEL_CONTROL_SERVER)
        /* LevelControl Cluster - Server */
        tsCLD_LevelControl sLevelControlServerCluster;
        tsCLD_LevelControlCustomDataStructure
                           sLevelControlServerCustomDataStructure;
    #endif

    #if (defined CLD_COLOUR_CONTROL) && (defined COLOUR_CONTROL_SERVER)
```

```
        /* Colour Control Cluster - Server */
        tsCLD_ColourControl sColourControlServerCluster;
        tsCLD_ColourControlCustomDataStructure
                            sColourControlServerCustomDataStructure;
    #endif

    #if (defined CLD_ZLL_COMMISSION) && (defined ZLL_COMMISSION_SERVER)
        tsCLD_ZllCommission sZllCommissionServerCluster;
        tsCLD_ZllCommissionCustomDataStructure
                            sZllCommissionServerCustomDataStructure;
    #endif

    #if (defined CLD_OTA) && (defined OTA_CLIENT)
        /* OTA cluster - Client */
        tsCLD_AS_Ota sCLD_OTA;
        tsOTA_Common sCLD_OTA_CustomDataStruct;
    #endif
} tsZLO_ColourTemperatureLightDevice;
```

3.13.3 Registration Function

The following **eZLO_RegisterColourTemperatureLightEndPoint()** function is the endpoint registration function for a Colour Temperature Light device.

teZCL_Status
eZLO_RegisterColourTemperatureLightEndPoint(
  uint8 *u8EndPointIdentifier*,
  tfpZCL_ZCLCallBackFunction *cbCallBack*,
  tsZLO_ColourTemperatureLightDevice **psDeviceInfo*);

Description

This function is used to register an endpoint which will support a Colour Temperature Light device. The function must be called after **eZCL_Initialise()**.

The specified identifier for the endpoint is a number in the range 1 to 240 (endpoint 0 is reserved for ZigBee use). Application endpoints are normally numbered consecutively starting at 1. The specified number must be less than or equal to the value of ZLO_NUMBER_OF_ENDPOINTS defined in the **zcl_options.h** file, which represents the highest endpoint number used for applications.

As part of this function call, you must specify a user-defined callback function that will be invoked when an event occurs that is associated with the endpoint. This callback function is defined according to the typedef:

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

You must also provide a pointer to a `tsZLO_ColourTemperatureLightDevice` structure (see Section 3.13.2) which will be used to store all variables relating to the Colour Temperature Light device associated with the endpoint. The `sEndPoint` and `sClusterInstance` fields of this structure are set by this function and must not be directly written to by the application.

The function may be called multiple times if more than one endpoint is being used - for example, if more than one Colour Temperature Light device is housed in the same hardware, sharing the same module.

Parameters

u8EndPointIdentifier	Endpoint that is to be associated with the registered structure and callback function
cbCallBack	Pointer to the function that will be used to indicate events to the application for this endpoint
psDeviceInfo	Pointer to the structure that will act as storage for all variables related to the device being registered on this endpoint (see Section 3.13.2). The `sEndPoint` and `sClusterInstance` fields are set by this register function for internal use and must not be written to by the application

Returns

E_ZCL_SUCCESS
E_ZCL_FAIL
E_ZCL_ERR_PARAMETER_NULL
E_ZCL_ERR_PARAMETER_RANGE
E_ZCL_ERR_EP_RANGE
E_ZCL_ERR_CLUSTER_0
E_ZCL_ERR_CALLBACK_NULL
E_ZCL_ERR_CLUSTER_NULL
E_ZCL_ERR_SECURITY_RANGE
E_ZCL_ERR_CLUSTER_ID_RANGE
E_ZCL_ERR_MANUFACTURER_SPECIFIC
E_ZCL_ERR_ATTRIBUTE_TYPE_UNSUPPORTED
E_ZCL_ERR_ATTRIBUTE_ID_ORDER
E_ZCL_ERR_ATTRIBUTES_ACCESS

The above codes are described in the *ZCL User Guide (JN-UG-3132)*.

3.14 Extended Colour Light

The Extended Colour Light device is typically used in nodes that contain a colour lamp with adjustable colour and brightness. This device supports a range of colour parameters, including hue/saturation, enhanced hue, colour temperature, colour loop and XY.

- The Device ID is 0x010D
- The header file for the device is **extended_colour_light.h**
- The clusters supported by the device are listed in Section 3.14.1
- The device structure, `tsZLO_ExtendedColourLightDevice`, is listed in Section 3.14.2
- The endpoint registration function for the device, **eZLO_RegisterExtendedColourLightEndPoint()**, is detailed in Section 3.14.3

3.14.1 Supported Clusters

The clusters supported by the Extended Colour Light device are as follows:

Server (Input) Side	Client (Output) Side
Mandatory	
Basic	
Identify	
Groups	
Scenes	
On/Off	
Level Control	
Colour Control	
Optional	
Touchlink Commissioning	OTA Upgrade

Table 15: Clusters for Extended Colour Light

The mandatory attributes within each cluster for this device type are indicated in the *ZigBee Lighting and Occupancy Device Specification (15-0014-01)*.

3.14.2 Device Structure

The following `tsZLO_ExtendedColourLightDevice` structure is the shared structure for a Extended Colour Light device:

```
typedef struct
{
    tsZCL_EndPointDefinition sEndPoint;

    /* Cluster instances */
    tsZLO_ExtendedColourLightDeviceClusterInstances
                                     sClusterInstance;

    #if (defined CLD_BASIC) && (defined BASIC_SERVER)
        /* Basic Cluster - Server */
        tsCLD_Basic sBasicServerCluster;
    #endif

    #if (defined CLD_IDENTIFY) && (defined IDENTIFY_SERVER)
        /* Identify Cluster - Server */
        tsCLD_Identify sIdentifyServerCluster;
        tsCLD_IdentifyCustomDataStructure
                                     sIdentifyServerCustomDataStructure;
    #endif

    #if (defined CLD_ONOFF) && (defined ONOFF_SERVER)
        /* On/Off Cluster - Server */
        tsCLD_OnOff sOnOffServerCluster;
        tsCLD_OnOffCustomDataStructure sOnOffServerCustomDataStructure;
    #endif

    #if (defined CLD_GROUPS) && (defined GROUPS_SERVER)
        /* Groups Cluster - Server */
        tsCLD_Groups sGroupsServerCluster;
        tsCLD_GroupsCustomDataStructure sGroupsServerCustomDataStructure;
    #endif

    #if (defined CLD_SCENES) && (defined SCENES_SERVER)
        /* Scenes Cluster - Server */
        tsCLD_Scenes sScenesServerCluster;
        tsCLD_ScenesCustomDataStructure sScenesServerCustomDataStructure;
    #endif

    #if (defined CLD_LEVEL_CONTROL) && (defined LEVEL_CONTROL_SERVER)
        /* LevelControl Cluster - Server */
        tsCLD_LevelControl sLevelControlServerCluster;
        tsCLD_LevelControlCustomDataStructure
                               sLevelControlServerCustomDataStructure;
    #endif
    #if (defined CLD_COLOUR_CONTROL) && (defined COLOUR_CONTROL_SERVER)
        /* Colour Control Cluster - Server */
        tsCLD_ColourControl sColourControlServerCluster;
        tsCLD_ColourControlCustomDataStructure
                            sColourControlServerCustomDataStructure;
    #endif

    #if (defined CLD_ZLL_COMMISSION) && (defined ZLL_COMMISSION_SERVER)
        tsCLD_ZllCommission sZllCommissionServerCluster;
        tsCLD_ZllCommissionCustomDataStructure
                            sZllCommissionServerCustomDataStructure;
    #endif

    #if (defined CLD_OTA) && (defined OTA_CLIENT)
        /* OTA cluster - Client */
        tsCLD_AS_Ota sCLD_OTA;
        tsOTA_Common sCLD_OTA_CustomDataStruct;
    #endif
} tsZLO_ExtendedColourLightDevice;
```

3.14.3 Registration Function

The following **eZLO_RegisterExtendedColourLightEndPoint()** function is the endpoint registration function for an Extended Colour Light device.

teZCL_Status eZLO_RegisterExtendedColourLightEndPoint(
uint8 *u8EndPointIdentifier*,
tfpZCL_ZCLCallBackFunction *cbCallBack*,
tsZLO_ExtendedColourLightDevice **psDeviceInfo*);

Description

This function is used to register an endpoint which will support an Extended Colour Light device. The function must be called after **eZCL_Initialise()**.

The specified identifier for the endpoint is a number in the range 1 to 240 (endpoint 0 is reserved for ZigBee use). Application endpoints are normally numbered consecutively starting at 1. The specified number must be less than or equal to the value of ZLO_NUMBER_OF_ENDPOINTS defined in the **zcl_options.h** file, which represents the highest endpoint number used for applications.

As part of this function call, you must specify a user-defined callback function that will be invoked when an event occurs that is associated with the endpoint. This callback function is defined according to the typedef:

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

You must also provide a pointer to a `tsZLO_ExtendedColourLightDevice` structure (see Section 3.14.2) which will be used to store all variables relating to the Extended Colour Light device associated with the endpoint. The `sEndPoint` and `sClusterInstance` fields of this structure are set by this function and must not be directly written to by the application.

The function may be called multiple times if more than one endpoint is being used - for example, if more than one Extended Colour Light device is housed in the same hardware, sharing the same module.

Parameters

u8EndPointIdentifier	Endpoint that is to be associated with the registered structure and callback function
cbCallBack	Pointer to the function that will be used to indicate events to the application for this endpoint
psDeviceInfo	Pointer to the structure that will act as storage for all variables related to the device being registered on this endpoint (see Section 3.14.2). The `sEndPoint` and `sClusterInstance` fields are set by this register function for internal use and must not be written to by the application

Returns

E_ZCL_SUCCESS
E_ZCL_FAIL
E_ZCL_ERR_PARAMETER_NULL
E_ZCL_ERR_PARAMETER_RANGE
E_ZCL_ERR_EP_RANGE
E_ZCL_ERR_CLUSTER_0
E_ZCL_ERR_CALLBACK_NULL
E_ZCL_ERR_CLUSTER_NULL
E_ZCL_ERR_SECURITY_RANGE
E_ZCL_ERR_CLUSTER_ID_RANGE
E_ZCL_ERR_MANUFACTURER_SPECIFIC
E_ZCL_ERR_ATTRIBUTE_TYPE_UNSUPPORTED
E_ZCL_ERR_ATTRIBUTE_ID_ORDER
E_ZCL_ERR_ATTRIBUTES_ACCESS

The above codes are described in the *ZCL User Guide (JN-UG-3132)*.

3.15 Light Level Sensor

The Light Level Sensor device measures the illumination level in an area and can be used to switch on/off a lighting device, such as an On/Off Ballast.

- The Device ID is 0x010E
- The header file for the device is **light_level_sensor.h**
- The clusters supported by the device are listed in Section 3.15.1
- The device structure, `tsZLO_LightLevelSensorDevice`, is listed in Section 3.15.2
- The endpoint registration function for the device, **eZLO_RegisterLightLevelSensorEndPoint()**, is detailed in Section 3.15.3

3.15.1 Supported Clusters

The clusters used by the Light Level Sensor device are listed in the table below.

Server (Input) Side	Client (Output) Side
Mandatory	
Basic	Identify
Identify	
Illuminance Level Sensing	
Optional	
	OTA Upgrade
	Groups

Table 16: Clusters for Light Level Sensor

3.15.2 Device Structure

The following `tsZLO_LightLevelSensorDevice` structure is the shared structure for a Light Level Sensor device:

```
typedef struct
{
    tsZCL_EndPointDefinition sEndPoint;

    /* Cluster instances */
    tsZLO_LightLevelSensorDeviceClusterInstances sClusterInstance;

    /* Mandatory server clusters */
    #if (defined CLD_BASIC) && (defined BASIC_SERVER)
        /* Basic Cluster - Server */
        tsCLD_Basic sBasicServerCluster;
    #endif

    #if (defined CLD_IDENTIFY) && (defined IDENTIFY_SERVER)
        /* Identify Cluster - Server */
        tsCLD_Identify sIdentifyServerCluster;
        tsCLD_IdentifyCustomDataStructure
                       sIdentifyServerCustomDataStructure;
    #endif

    #if (defined CLD_ILLUMINANCE_LEVEL_SENSING) && (defined
ILLUMINANCE_LEVEL_SENSING_SERVER)
        tsCLD_IlluminanceLevelSensing
                       sIlluminanceLevelSensingServerCluster;
    #endif

    /* Optional server clusters */
    #if (defined CLD_POLL_CONTROL) && (defined POLL_CONTROL_SERVER)
        tsCLD_PollControl sPollControlServerCluster;
        tsCLD_PollControlCustomDataStructure
                          sPollControlServerCustomDataStructure;
    #endif

    /* Mandatory server clusters */
    #if (defined CLD_IDENTIFY) && (defined IDENTIFY_CLIENT)
        /* Identify Cluster - Client */
        tsCLD_Identify sIdentifyClientCluster;
        tsCLD_IdentifyCustomDataStructure
                       sIdentifyClientCustomDataStructure;
    #endif

    /* Recommended Optional client clusters */
    #if (defined CLD_GROUPS) && (defined GROUPS_CLIENT)
        /* Groups Cluster - Client */
        tsCLD_Groups sGroupsClientCluster;
        tsCLD_GroupsCustomDataStructure sGroupsClientCustomDataStructure;
    #endif

    #if (defined CLD_OTA) && (defined OTA_CLIENT)
        tsCLD_AS_Ota sCLD_OTA;
        tsOTA_Common sCLD_OTA_CustomDataStruct;
    #endif

} tsZLO_LightLevelSensorDevice;
```

3.15.3 Registration Function

The following **eZLO_RegisterLightLevelSensorEndPoint()** function is the endpoint registration function for a Light Level Sensor device.

> **teZCL_Status eZLO_RegisterLightLevelSensorEndPoint(**
> **uint8** *u8EndPointIdentifier*,
> **tfpZCL_ZCLCallBackFunction** *cbCallBack*,
> **tsZLO_LightLevelSensorDevice** **psDeviceInfo*);

Description

This function is used to register an endpoint which will support a Light Level Sensor device. The function must be called after **eZCL_Initialise()**.

The specified identifier for the endpoint is a number in the range 1 to 240 (endpoint 0 is reserved for ZigBee use). Application endpoints are normally numbered consecutively starting at 1. The specified number must be less than or equal to the value of ZLO_NUMBER_OF_ENDPOINTS defined in the **zcl_options.h** file, which represents the highest endpoint number used for applications.

As part of this function call, you must specify a user-defined callback function that will be invoked when an event occurs that is associated with the endpoint. This callback function is defined according to the typedef:

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

You must also provide a pointer to a `tsZLO_LightLevelSensorDevice` structure (see Section 3.16.2) which will be used to store all variables relating to the Light Level Sensor device associated with the endpoint. The `sEndPoint` and `sClusterInstance` fields of this structure are set by this function and must not be directly written to by the application.

The function may be called multiple times if more than one endpoint is being used - for example, if more than one Light Level Sensor device is housed in the same hardware, sharing the same module.

Parameters

u8EndPointIdentifier	Endpoint that is to be associated with the registered structure and callback function
cbCallBack	Pointer to the function that will be used to indicate events to the application for this endpoint
psDeviceInfo	Pointer to the structure that will act as storage for all variables related to the device being registered on this endpoint (see Section 3.16.2). The `sEndPoint` and `sClusterInstance` fields are set by this register function for internal use and must not be written to by the application

Returns

E_ZCL_SUCCESS
E_ZCL_FAIL
E_ZCL_ERR_PARAMETER_NULL
E_ZCL_ERR_PARAMETER_RANGE
E_ZCL_ERR_EP_RANGE
E_ZCL_ERR_CLUSTER_0
E_ZCL_ERR_CALLBACK_NULL
E_ZCL_ERR_CLUSTER_NULL
E_ZCL_ERR_SECURITY_RANGE
E_ZCL_ERR_CLUSTER_ID_RANGE
E_ZCL_ERR_MANUFACTURER_SPECIFIC
E_ZCL_ERR_ATTRIBUTE_TYPE_UNSUPPORTED
E_ZCL_ERR_ATTRIBUTE_ID_ORDER
E_ZCL_ERR_ATTRIBUTES_ACCESS

The above codes are described in the *ZCL User Guide (JN-UG-3132)*.

3.16 Colour Controller

The Colour Controller device is typically used in a node that issues colour-control commands to adjust the intensity or colour of a lighting device, or switch it on/off.

- The Device ID is 0x0800
- The header file for the device is **colour_controller.h**
- The clusters supported by the device are listed in Section 3.16.1
- The device structure, `tsZLO_ColourControllerDevice`, is listed in Section 3.16.2
- The endpoint registration function for the device, **eZLO_RegisterColourRemoteEndPoint()**, is detailed in Section 3.16.3

3.16.1 Supported Clusters

The clusters supported by the Colour Controller device are as follows:

Server (Input) Side	Client (Output) Side
Mandatory	
Basic	On/Off
Identify	Identify
	Level Control
	Colour Control
Optional	
Touchlink Commissioning	Touchlink Commissioning
	Groups
	OTA Upgrade

Table 17: Clusters for Colour Controller

The mandatory attributes within each cluster for this device type are indicated in the *ZigBee Lighting and Occupancy Device Specification (15-0014-01)*.

3.16.2 Device Structure

The following `tsZLO_ColourControllerDevice` structure is the shared structure for a Colour Controller device:

```
typedef struct
{
    tsZCL_EndPointDefinition sEndPoint;

    /* Cluster instances */
    tsZLO_ColourControllerDeviceClusterInstances sClusterInstance;

    #if (defined CLD_BASIC) && (defined BASIC_SERVER)
        /* Basic Cluster - Server */
        tsCLD_Basic sBasicServerCluster;
    #endif

    #if (defined CLD_IDENTIFY) && (defined IDENTIFY_SERVER)
        /* Identify Cluster - Server */
        tsCLD_Identify sIdentifyServerCluster;
        tsCLD_IdentifyCustomDataStructure
                                sIdentifyServerCustomDataStructure;
    #endif

    #if (defined CLD_ZLL_COMMISSION) && (defined ZLL_COMMISSION_SERVER)
        tsCLD_ZllCommission sZllCommissionServerCluster;
        tsCLD_ZllCommissionCustomDataStructure
                            sZllCommissionServerCustomDataStructure;
    #endif

        /* Mandatory client clusters */
    #if (defined CLD_IDENTIFY) && (defined IDENTIFY_CLIENT)
        /* Identify Cluster - Client */
        tsCLD_Identify sIdentifyClientCluster;
        tsCLD_IdentifyCustomDataStructure
                       sIdentifyClientCustomDataStructure;
    #endif

    #if (defined CLD_BASIC) && (defined BASIC_CLIENT)
        /* Basic Cluster - Client */
        tsCLD_Basic sBasicClientCluster;
    #endif

    #if (defined CLD_ONOFF) && (defined ONOFF_CLIENT)
        /* On/Off Cluster - Client */
        tsCLD_OnOff sOnOffClientCluster;
    #endif

    #if (defined CLD_LEVEL_CONTROL) && (defined LEVEL_CONTROL_CLIENT)
        /* Level Control Cluster - Client */
```

```
            tsCLD_LevelControl sLevelControlClientCluster;
            tsCLD_LevelControlCustomDataStructure
                               sLevelControlClientCustomDataStructure;
        #endif

        #if (defined CLD_COLOUR_CONTROL) && (defined COLOUR_CONTROL_CLIENT)
            /* Colour Control Cluster - Client */
            tsCLD_ColourControl sColourControlClientCluster;
            tsCLD_ColourControlCustomDataStructure
                                sColourControlClientCustomDataStructure;
        #endif

        /* Optional client cluster */
        #if (defined CLD_GROUPS) && (defined GROUPS_CLIENT)
            /* Groups Cluster - Client */
            tsCLD_Groups sGroupsClientCluster;
            tsCLD_GroupsCustomDataStructure sGroupsClientCustomDataStructure;
        #endif

        #if (defined CLD_OTA) && (defined OTA_CLIENT)
            /* OTA cluster - Client */
            tsCLD_AS_Ota sCLD_OTA;
            tsOTA_Common sCLD_OTA_CustomDataStruct;
        #endif

        #if (defined CLD_ZLL_COMMISSION) && (defined ZLL_COMMISSION_CLIENT)
            tsCLD_ZllCommission sZllCommissionClientCluster;
            tsCLD_ZllCommissionCustomDataStructure
                                sZllCommissionClientCustomDataStructure;
        #endif

    } tsZLO_ColourControllerDevice;
```

3.16.3 Registration Function

The following **eZLO_RegisterColourControllerEndPoint()** function is the endpoint registration function for a Colour Controller device.

teZCL_Status eZLO_RegisterColourControllerEndPoint(
uint8 *u8EndPointIdentifier*,
tfpZCL_ZCLCallBackFunction *cbCallBack*,
tsZLO_ColourControllerDevice **psDeviceInfo*);

Description

This function is used to register an endpoint which will support a Colour Controller device. The function must be called after **eZCL_Initialise()**.

The specified identifier for the endpoint is a number in the range 1 to 240 (endpoint 0 is reserved for ZigBee use). Application endpoints are normally numbered consecutively starting at 1. The specified number must be less than or equal to the value of ZLO_NUMBER_OF_ENDPOINTS defined in the **zcl_options.h** file, which represents the highest endpoint number used for applications.

As part of this function call, you must specify a user-defined callback function that will be invoked when an event occurs that is associated with the endpoint. This callback function is defined according to the typedef:

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

You must also provide a pointer to a `tsZLO_ColourControllerDevice` structure (see Section 3.16.2) which will be used to store all variables relating to the Colour Controller device associated with the endpoint. The `sEndPoint` and `sClusterInstance` fields of this structure are set by this function and must not be directly written to by the application.

The function may be called multiple times if more than one endpoint is being used - for example, if more than one Colour Controller device is housed in the same hardware, sharing the same module.

Parameters

u8EndPointIdentifier	Endpoint that is to be associated with the registered structure and callback function
cbCallBack	Pointer to the function that will be used to indicate events to the application for this endpoint
psDeviceInfo	Pointer to the structure that will act as storage for all variables related to the device being registered on this endpoint (see Section 3.16.2). The `sEndPoint` and `sClusterInstance` fields are set by this register function for internal use and must not be written to by the application

Returns

E_ZCL_SUCCESS
E_ZCL_FAIL
E_ZCL_ERR_PARAMETER_NULL
E_ZCL_ERR_PARAMETER_RANGE
E_ZCL_ERR_EP_RANGE
E_ZCL_ERR_CLUSTER_0
E_ZCL_ERR_CALLBACK_NULL
E_ZCL_ERR_CLUSTER_NULL
E_ZCL_ERR_SECURITY_RANGE
E_ZCL_ERR_CLUSTER_ID_RANGE
E_ZCL_ERR_MANUFACTURER_SPECIFIC
E_ZCL_ERR_ATTRIBUTE_TYPE_UNSUPPORTED
E_ZCL_ERR_ATTRIBUTE_ID_ORDER
E_ZCL_ERR_ATTRIBUTES_ACCESS

The above codes are described in the *ZCL User Guide (JN-UG-3132)*.

3.17 Colour Scene Controller

The Colour Scene Controller device is typically used in nodes that support scenes and that issue colour-control commands (to adjust the intensity or colour of a lighting device, or switch it on/off) - for example, to control a Colour Dimmable Light.

- The Device ID is 0x0810
- The header file for the device is **colour_scene_controller.h**
- The clusters supported by the device are listed in Section 3.17.1
- The device structure, `tsZLO_ColourSceneControllerDevice`, is listed in Section 3.17.2
- The endpoint registration function for the device, **eZLO_RegisterColourSceneControllerEndPoint()**, is detailed in Section 3.17.3

3.17.1 Supported Clusters

The clusters supported by the Colour Scene Controller device are as follows:

Server (Input) Side	Client (Output) Side
Mandatory	
Basic	On/Off
Identify	Identify
	Level Control
	Colour Control
	Scenes
Optional	
Touchlink Commissioning	Touchlink Commissioning
	Groups
	OTA Upgrade

Table 18: Clusters for Colour Scene Controller

The mandatory attributes within each cluster for this device type are indicated in the *ZigBee Lighting and Occupancy Device Specification (15-0014-01)*.

3.17.2 Device Structure

The following `tsZLO_ColourSceneControllerDevice` structure is the shared structure for a Colour Scene Controller device:

```
typedef struct
{
    tsZCL_EndPointDefinition sEndPoint;

    /* Cluster instances */
    tsZLO_ColourSceneControllerDeviceClusterInstances sClusterInstance;

    #if (defined CLD_BASIC) && (defined BASIC_SERVER)
        /* Basic Cluster - Server */
        tsCLD_Basic sBasicServerCluster;
    #endif

    #if (defined CLD_IDENTIFY) && (defined IDENTIFY_SERVER)
        /* Identify Cluster - Server */
        tsCLD_Identify sIdentifyServerCluster;
        tsCLD_IdentifyCustomDataStructure
                       sIdentifyServerCustomDataStructure;
    #endif

    #if (defined CLD_ZLL_COMMISSION) && (defined ZLL_COMMISSION_SERVER)
        tsCLD_ZllCommission sZllCommissionServerCluster;
        tsCLD_ZllCommissionCustomDataStructure
                            sZllCommissionServerCustomDataStructure;
    #endif

        /* Mandatory client clusters */
    #if (defined CLD_IDENTIFY) && (defined IDENTIFY_CLIENT)
        /* Identify Cluster - Client */
        tsCLD_Identify sIdentifyClientCluster;
        tsCLD_IdentifyCustomDataStructure
                       sIdentifyClientCustomDataStructure;
    #endif

    #if (defined CLD_BASIC) && (defined BASIC_CLIENT)
        /* Basic Cluster - Server */
        tsCLD_Basic sBasicClientCluster;
    #endif

    #if (defined CLD_ONOFF) && (defined ONOFF_CLIENT)
        /* On/Off Cluster - Client */
        tsCLD_OnOff sOnOffClientCluster;
    #endif

    #if (defined CLD_LEVEL_CONTROL) && (defined LEVEL_CONTROL_CLIENT)
        /* Level Control Cluster - Client */
        tsCLD_LevelControl sLevelControlClientCluster;
        tsCLD_LevelControlCustomDataStructure
                           sLevelControlClientCustomDataStructure;
    #endif

    #if (defined CLD_COLOUR_CONTROL) && (defined COLOUR_CONTROL_CLIENT)
        /* Colour Control Cluster - Client */
        tsCLD_ColourControl sColourControlClientCluster;
        tsCLD_ColourControlCustomDataStructure
                            sColourControlClientCustomDataStructure;
    #endif

    #if (defined CLD_SCENES) && (defined SCENES_CLIENT)
        /* Scenes Cluster - Client */
        tsCLD_Scenes sScenesClientCluster;
        tsCLD_ScenesCustomDataStructure sScenesClientCustomDataStructure;
    #endif

    /* Recommended Optional Client Cluster */
    #if (defined CLD_GROUPS) && (defined GROUPS_CLIENT)
        /* Groups Cluster - Client */
        tsCLD_Groups sGroupsClientCluster;
        tsCLD_GroupsCustomDataStructure sGroupsClientCustomDataStructure;
    #endif

    #if (defined CLD_OTA) && (defined OTA_CLIENT)
        /* OTA cluster - Client */
        tsCLD_AS_Ota sCLD_OTA;
        tsOTA_Common sCLD_OTA_CustomDataStruct;
    #endif

    #if (defined CLD_ZLL_COMMISSION) && (defined ZLL_COMMISSION_CLIENT)
        tsCLD_ZllCommission sZllCommissionClientCluster;
        tsCLD_ZllCommissionCustomDataStructure
                            sZllCommissionClientCustomDataStructure;
    #endif
} tsZLO_ColourSceneControllerDevice;
```

3.17.3 Registration Function

The following **eZLO_RegisterColourSceneControllerEndPoint()** function is the endpoint registration function for a Colour Scene Controller device.

teZCL_Status
eZLO_RegisterColourSceneControllerEndPoint(
    uint8 *u8EndPointIdentifier*,
    tfpZCL_ZCLCallBackFunction *cbCallBack*,
    tsZLO_ColourSceneControllerDevice **psDeviceInfo***);**

Description

This function is used to register an endpoint which will support a Colour Scene Controller device. The function must be called after **eZCL_Initialise()**.

The specified identifier for the endpoint is a number in the range 1 to 240 (endpoint 0 is reserved for ZigBee use). Application endpoints are normally numbered consecutively starting at 1. The specified number must be less than or equal to the value of ZLO_NUMBER_OF_ENDPOINTS defined in the **zcl_options.h** file, which represents the highest endpoint number used for applications.

As part of this function call, you must specify a user-defined callback function that will be invoked when an event occurs that is associated with the endpoint. This callback function is defined according to the typedef:

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

You must also provide a pointer to a `tsZLO_ColourSceneControllerDevice` structure (see Section 3.17.2) which will be used to store all variables relating to the Colour Scene Controller device associated with the endpoint. The `sEndPoint` and `sClusterInstance` fields of this structure are set by this function and must not be directly written to by the application.

The function may be called multiple times if more than one endpoint is being used - for example, if more than one Colour Scene Controller device is housed in the same hardware, sharing the same module.

Parameters

u8EndPointIdentifier	Endpoint that is to be associated with the registered structure and callback function
cbCallBack	Pointer to the function that will be used to indicate events to the application for this endpoint
psDeviceInfo	Pointer to the structure that will act as storage for all variables related to the device being registered on this endpoint (see Section 3.17.2). The `sEndPoint` and `sClusterInstance` fields are set by this register function for internal use and must not be written to by the application

Returns

E_ZCL_SUCCESS
E_ZCL_FAIL
E_ZCL_ERR_PARAMETER_NULL
E_ZCL_ERR_PARAMETER_RANGE
E_ZCL_ERR_EP_RANGE
E_ZCL_ERR_CLUSTER_0
E_ZCL_ERR_CALLBACK_NULL
E_ZCL_ERR_CLUSTER_NULL
E_ZCL_ERR_SECURITY_RANGE
E_ZCL_ERR_CLUSTER_ID_RANGE
E_ZCL_ERR_MANUFACTURER_SPECIFIC
E_ZCL_ERR_ATTRIBUTE_TYPE_UNSUPPORTED
E_ZCL_ERR_ATTRIBUTE_ID_ORDER
E_ZCL_ERR_ATTRIBUTES_ACCESS

The above codes are described in the *ZCL User Guide (JN-UG-3132)*.

3.18 Non-Colour Controller

The Non-Colour Controller device is typically used in nodes that issue control commands that are not related to colour - for example, to control a Dimmable Light.

- The Device ID is 0x0820
- The header file for the device is **non_colour_controller.h**
- The clusters supported by the device are listed in Section 3.18.1
- The device structure, `tsZLO_NonColourControllerDevice`, is listed in Section 3.18.2
- The endpoint registration function for the device, **eZLO_RegisterNonColourControllerEndPoint()**, is detailed in Section 3.18.3

3.18.1 Supported Clusters

The clusters supported by the Non-Colour Controller device are as follows:

Server (Input) Side	Client (Output) Side
Mandatory	
Basic	On/Off
Identify	Identify
	Level Control
Optional	
Touchlink Commissioning	Touchlink Commissioning
	Groups
	OTA Upgrade

Table 19: Clusters for Non-Colour Controller

The mandatory attributes within each cluster for this device type are indicated in the *ZigBee Lighting and Occupancy Device Specification (15-0014-01)*.

3.18.2 Device Structure

The following `tsZLO_NonColourControllerDevice` structure is the shared structure for a Non-Colour Controller device:

```
typedef struct
{
    tsZCL_EndPointDefinition sEndPoint;

    /* Cluster instances */
    tsZLO_NonColourControllerDeviceClusterInstances sClusterInstance;

    #if (defined CLD_BASIC) && (defined BASIC_SERVER)
        /* Basic Cluster - Server */
        tsCLD_Basic sBasicServerCluster;
    #endif

    #if (defined CLD_IDENTIFY) && (defined IDENTIFY_SERVER)
        /* Identify Cluster - Server */
        tsCLD_Identify sIdentifyServerCluster;
       tsCLD_IdentifyCustomDataStructure sIdentifyServerCustomDataStructure;
    #endif

    #if (defined CLD_ZLL_COMMISSION) && (defined ZLL_COMMISSION_SERVER)
        tsCLD_ZllCommission sZllCommissionServerCluster;
        tsCLD_ZllCommissionCustomDataStructure
                            sZllCommissionServerCustomDataStructure;
    #endif

        /* Mandatory client clusters */
    #if (defined CLD_IDENTIFY) && (defined IDENTIFY_CLIENT)
        /* Identify Cluster - Client */
        tsCLD_Identify sIdentifyClientCluster;
       tsCLD_IdentifyCustomDataStructure sIdentifyClientCustomDataStructure;
    #endif

    #if (defined CLD_BASIC) && (defined BASIC_CLIENT)
        /* Basic Cluster - Client */
        tsCLD_Basic sBasicClientCluster;
    #endif

    #if (defined CLD_ONOFF) && (defined ONOFF_CLIENT)
        /* On/Off Cluster - Client */
        tsCLD_OnOff sOnOffClientCluster;
    #endif

    #if (defined CLD_LEVEL_CONTROL) && (defined LEVEL_CONTROL_CLIENT)
        /* Level Control Cluster - Client */
        tsCLD_LevelControl sLevelControlClientCluster;
        tsCLD_LevelControlCustomDataStructure
                           sLevelControlClientCustomDataStructure;
    #endif

    #if (defined CLD_GROUPS) && (defined GROUPS_CLIENT)
        /* Groups Cluster - Client */
        tsCLD_Groups sGroupsClientCluster;
        tsCLD_GroupsCustomDataStructure sGroupsClientCustomDataStructure;
    #endif

    #if (defined CLD_OTA) && (defined OTA_CLIENT)
        /* OTA cluster - Client */
        tsCLD_AS_Ota sCLD_OTA;
        tsOTA_Common sCLD_OTA_CustomDataStruct;
    #endif

    #if (defined CLD_ZLL_COMMISSION) && (defined ZLL_COMMISSION_CLIENT)
        tsCLD_ZllCommission sZllCommissionClientCluster;
        tsCLD_ZllCommissionCustomDataStructure
                            sZllCommissionClientCustomDataStructure;
    #endif

} tsZLO_NonColourControllerDevice;
```

3.18.3 Registration Function

The following **eZLO_RegisterNonColourControllerEndPoint()** function is the endpoint registration function for a Non-Colour Controller device.

teZCL_Status eZLO_RegisterNonColourControllerEndPoint(
uint8 *u8EndPointIdentifier*,
tfpZCL_ZCLCallBackFunction *cbCallBack*,
tsZLO_NonColourControllerDevice **psDeviceInfo*);

Description

This function is used to register an endpoint which will support a Non-Colour Controller device. The function must be called after **eZCL_Initialise()**.

The specified identifier for the endpoint is a number in the range 1 to 240 (endpoint 0 is reserved for ZigBee use). Application endpoints are normally numbered consecutively starting at 1. The specified number must be less than or equal to the value of ZLO_NUMBER_OF_ENDPOINTS defined in the **zcl_options.h** file, which represents the highest endpoint number used for applications.

As part of this function call, you must specify a user-defined callback function that will be invoked when an event occurs that is associated with the endpoint. This callback function is defined according to the typedef:

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

You must also provide a pointer to a `tsZLO_NonColourControllerDevice` structure (see Section 3.18.2) which will be used to store all variables relating to the Non-Colour Controller device associated with the endpoint. The `sEndPoint` and `sClusterInstance` fields of this structure are set by this function and must not be directly written to by the application.

The function may be called multiple times if more than one endpoint is being used - for example, if more than one Non-Colour Controller device is housed in the same hardware, sharing the same module.

Parameters

u8EndPointIdentifier	Endpoint that is to be associated with the registered structure and callback function
cbCallBack	Pointer to the function that will be used to indicate events to the application for this endpoint
psDeviceInfo	Pointer to the structure that will act as storage for all variables related to the device being registered on this endpoint (see Section 3.18.2). The `sEndPoint` and `sClusterInstance` fields are set by this register function for internal use and must not be written to by the application

Returns

E_ZCL_SUCCESS
E_ZCL_FAIL
E_ZCL_ERR_PARAMETER_NULL
E_ZCL_ERR_PARAMETER_RANGE
E_ZCL_ERR_EP_RANGE
E_ZCL_ERR_CLUSTER_0
E_ZCL_ERR_CALLBACK_NULL
E_ZCL_ERR_CLUSTER_NULL
E_ZCL_ERR_SECURITY_RANGE
E_ZCL_ERR_CLUSTER_ID_RANGE
E_ZCL_ERR_MANUFACTURER_SPECIFIC
E_ZCL_ERR_ATTRIBUTE_TYPE_UNSUPPORTED
E_ZCL_ERR_ATTRIBUTE_ID_ORDER
E_ZCL_ERR_ATTRIBUTES_ACCESS

The above codes are described in the *ZCL User Guide (JN-UG-3132)*.

3.19 Non-Colour Scene Controller

The Non-Colour Scene Controller device is typically used in nodes that support ‘scenes’ and that issue control commands which are not related to colour - for example, to control a Dimmable Light.

- The Device ID is 0x0830
- The header file for the device is **non_colour_scene_controller.h**
- The clusters supported by the device are listed in Section 3.19.1
- The device structure, `tsZLO_NonColourSceneRemoteDevice`, is listed in Section 3.19.2
- The endpoint registration function for the device, **eZLO_RegisterNonColourSceneControllerEndPoint()**, is detailed in Section 3.19.3

3.19.1 Supported Clusters

The clusters supported by the Non-Colour Scene Controller device are as follows:

Server (Input) Side	**Client (Output) Side**
Mandatory	
Basic	On/Off
Identify	Identify
	Level Control
	Scenes
Optional	
Touchlink Commissioning	Touchlink Commissioning
	Groups
	OTA Upgrade

Table 20: Clusters for Non-Colour Scene Controller

The mandatory attributes within each cluster for this device type are indicated in the *ZigBee Lighting and Occupancy Device Specification (15-0014-01)*.

3.19.2 Device Structure

The following `tsZLO_NonColourSceneControllerDevice` structure is the shared structure for a Non-Colour Scene Controller device:

```
typedef struct
{
    tsZCL_EndPointDefinition sEndPoint;

    /* Cluster instances */
    tsZLO_NonColourSceneControllerDeviceClusterInstances sClusterInstance;

    #if (defined CLD_BASIC) && (defined BASIC_SERVER)
        /* Basic Cluster - Server */
        tsCLD_Basic sBasicServerCluster;
    #endif

    #if (defined CLD_IDENTIFY) && (defined IDENTIFY_SERVER)
        /* Identify Cluster - Server */
        tsCLD_Identify sIdentifyServerCluster;
        tsCLD_IdentifyCustomDataStructure
                       sIdentifyServerCustomDataStructure;
    #endif

    #if (defined CLD_ZLL_COMMISSION) && (defined ZLL_COMMISSION_SERVER)
        tsCLD_ZllCommission sZllCommissionServerCluster;
        tsCLD_ZllCommissionCustomDataStructure
                            sZllCommissionServerCustomDataStructure;
    #endif

        /* Mandatory client clusters */
    #if (defined CLD_IDENTIFY) && (defined IDENTIFY_CLIENT)
        /* Identify Cluster - Client */
        tsCLD_Identify sIdentifyClientCluster;
        tsCLD_IdentifyCustomDataStructure
                       sIdentifyClientCustomDataStructure;
    #endif

    #if (defined CLD_BASIC) && (defined BASIC_CLIENT)
        /* Basic Cluster - Client */
        tsCLD_Basic sBasicClientCluster;
    #endif

    #if (defined CLD_ONOFF) && (defined ONOFF_CLIENT)
        /* On/Off Cluster - Client */
        tsCLD_OnOff sOnOffClientCluster;
    #endif

    #if (defined CLD_LEVEL_CONTROL) && (defined LEVEL_CONTROL_CLIENT)
        /* Level Control Cluster - Client */
        tsCLD_LevelControl sLevelControlClientCluster;
        tsCLD_LevelControlCustomDataStructure
                           sLevelControlClientCustomDataStructure;
    #endif

    #if (defined CLD_SCENES) && (defined SCENES_CLIENT)
        /* Scenes Cluster - Client */
        tsCLD_Scenes sScenesClientCluster;
        tsCLD_ScenesCustomDataStructure sScenesClientCustomDataStructure;
    #endif

    /* Recommended Optional Client Cluster */
    #if (defined CLD_GROUPS) && (defined GROUPS_CLIENT)
        /* Groups Cluster - Client */
        tsCLD_Groups sGroupsClientCluster;
        tsCLD_GroupsCustomDataStructure sGroupsClientCustomDataStructure;
    #endif

    #if (defined CLD_OTA) && (defined OTA_CLIENT)
        /* OTA cluster - Client */
        tsCLD_AS_Ota sCLD_OTA;
        tsOTA_Common sCLD_OTA_CustomDataStruct;
    #endif

    #if (defined CLD_ZLL_COMMISSION) && (defined ZLL_COMMISSION_CLIENT)
        tsCLD_ZllCommission sZllCommissionClientCluster;
        tsCLD_ZllCommissionCustomDataStructure
                            sZllCommissionClientCustomDataStructure;
    #endif
} tsZLO_NonColourSceneControllerDevice;
```

3.19.3 Registration Function

The following **eZLO_RegisterNonColourSceneControllerEndPoint()** function is the endpoint registration function for a Non-Colour Scene Controller device.

teZCL_Status
eZLO_RegisterNonColourSceneControllerEndPoint(
uint8 *u8EndPointIdentifier*,
tfpZCL_ZCLCallBackFunction *cbCallBack*,
tsZLO_NonColourSceneControllerDevice **psDeviceInfo*);

Description

This function is used to register an endpoint which will support a Non-Colour Scene Controller device. The function must be called after **eZCL_Initialise()**.

The specified identifier for the endpoint is a number in the range 1 to 240 (endpoint 0 is reserved for ZigBee use). Application endpoints are normally numbered consecutively starting at 1. The specified number must be less than or equal to the value of ZLO_NUMBER_OF_ENDPOINTS defined in the **zcl_options.h** file, which represents the highest endpoint number used for applications.

As part of this function call, you must specify a user-defined callback function that will be invoked when an event occurs that is associated with the endpoint. This callback function is defined according to the typedef:

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

You must also provide a pointer to a `tsZLO_NonColourSceneControllerDevice` structure (see Section 3.19.2) which will be used to store all variables relating to the Non-Colour Scene Controller device associated with the endpoint. The `sEndPoint` and `sClusterInstance` fields of this structure are set by this function and must not be directly written to by the application.

The function may be called multiple times if more than one endpoint is being used - for example, if more than one Non-Colour Scene Controller device is housed in the same hardware, sharing the same module.

Parameters

u8EndPointIdentifier	Endpoint that is to be associated with the registered structure and callback function
cbCallBack	Pointer to the function that will be used to indicate events to the application for this endpoint
psDeviceInfo	Pointer to the structure that will act as storage for all variables related to the device being registered on this endpoint (see Section 3.19.2). The `sEndPoint` and `sClusterInstance` fields are set by this register function for internal use and must not be written to by the application

Returns

E_ZCL_SUCCESS
E_ZCL_FAIL
E_ZCL_ERR_PARAMETER_NULL
E_ZCL_ERR_PARAMETER_RANGE
E_ZCL_ERR_EP_RANGE
E_ZCL_ERR_CLUSTER_0
E_ZCL_ERR_CALLBACK_NULL
E_ZCL_ERR_CLUSTER_NULL
E_ZCL_ERR_SECURITY_RANGE
E_ZCL_ERR_CLUSTER_ID_RANGE
E_ZCL_ERR_MANUFACTURER_SPECIFIC
E_ZCL_ERR_ATTRIBUTE_TYPE_UNSUPPORTED
E_ZCL_ERR_ATTRIBUTE_ID_ORDER
E_ZCL_ERR_ATTRIBUTES_ACCESS

The above codes are described in the *ZCL User Guide (JN-UG-3132)*.

3.20 Control Bridge

The Control Bridge device is typically used in nodes that relay control commands issued from another network, e.g. in an Internet router with a ZigBee interface.

- The Device ID is 0x0840
- The header file for the device is **control_bridge.h**
- The clusters supported by the device are listed in Section 3.20.1
- The device structure, `tsZLO_ControlBridgeDevice`, is listed in Section 3.20.2
- The endpoint registration function for the device, **eZLO_RegisterControlBridgeEndPoint()**, is detailed in Section 3.20.3

3.20.1 Supported Clusters

The clusters supported by the Control Bridge device are as follows:

Server (Input) Side	Client (Output) Side
Mandatory	
Basic	On/Off
Identify	Identify
	Groups
	Scenes
	Level Control
	Colour Control
Optional	
OTA Upgrade	OTA Upgrade
Touchlink Commissioning	Touchlink Commissioning
	Illuminance Measurement
	Illuminance Level Sensing
	Occupancy Sensing

Table 21: Clusters for Control Bridge

The mandatory attributes within each cluster for this device type are indicated in the *ZigBee Lighting and Occupancy Device Specification (15-0014-01)*.

3.20.2 Device Structure

The following `tsZLO_ControlBridgeDevice` structure is the shared structure for a Control Bridge device:

```
typedef struct
{
    tsZCL_EndPointDefinition sEndPoint;

    /* Cluster instances */
    tsZLO_ControlBridgeDeviceClusterInstances sClusterInstance;

    #if (defined CLD_BASIC) && (defined BASIC_SERVER)
        /* Basic Cluster - Server */
        tsCLD_Basic sBasicServerCluster;
    #endif

    #if (defined CLD_IDENTIFY) && (defined IDENTIFY_SERVER)
        /* Identify Cluster - Server */
        tsCLD_Identify sIdentifyServerCluster;
       tsCLD_IdentifyCustomDataStructure sIdentifyServerCustomDataStructure;
    #endif

    /* Recommended Optional Server Cluster */

    #if (defined CLD_OTA) && (defined OTA_SERVER)
    /* OTA cluster */
        tsCLD_AS_Ota sCLD_ServerOTA;
        tsOTA_Common sCLD_OTA_ServerCustomDataStruct;
    #endif

    #if (defined CLD_ZLL_COMMISSION) && (defined ZLL_COMMISSION_SERVER)
        tsCLD_ZllCommission sZllCommissionServerCluster;
        tsCLD_ZllCommissionCustomDataStructure
                            sZllCommissionServerCustomDataStructure;
    #endif

    /*
     * Mandatory client clusters
     */
    #if (defined CLD_IDENTIFY) && (defined IDENTIFY_CLIENT)
        /* Identify Cluster - Client */
        tsCLD_Identify sIdentifyClientCluster;
       tsCLD_IdentifyCustomDataStructure sIdentifyClientCustomDataStructure;
```

```
#endif

#if (defined CLD_BASIC) && (defined BASIC_CLIENT)
    /* Basic Cluster - Client */
    tsCLD_Basic sBasicClientCluster;
#endif

#if (defined CLD_ONOFF) && (defined ONOFF_CLIENT)
    /* On/Off Cluster - Client */
    tsCLD_OnOff sOnOffClientCluster;
#endif

#if (defined CLD_LEVEL_CONTROL) && (defined LEVEL_CONTROL_CLIENT)
    /* Level Control Cluster - Client */
    tsCLD_LevelControl sLevelControlClientCluster;
    tsCLD_LevelControlCustomDataStructure
                       sLevelControlClientCustomDataStructure;
#endif

#if (defined CLD_SCENES) && (defined SCENES_CLIENT)
    /* Scenes Cluster - Client */
    tsCLD_Scenes sScenesClientCluster;
    tsCLD_ScenesCustomDataStructure sScenesClientCustomDataStructure;
#endif

#if (defined CLD_GROUPS) && (defined GROUPS_CLIENT)
    /* Groups Cluster - Client */
    tsCLD_Groups sGroupsClientCluster;
    tsCLD_GroupsCustomDataStructure sGroupsClientCustomDataStructure;
#endif

#if (defined CLD_COLOUR_CONTROL) && (defined COLOUR_CONTROL_CLIENT)
    /* Colour Control Cluster - Client */
    tsCLD_ColourControl sColourControlClientCluster;
    tsCLD_ColourControlCustomDataStructure
                        sColourControlClientCustomDataStructure;
#endif

    /* Recommended Optional client clusters */

#if (defined CLD_OTA) && (defined OTA_CLIENT)
    /* OTA cluster */
    tsCLD_AS_Ota sCLD_OTA;
    tsOTA_Common sCLD_OTA_CustomDataStruct;
    #endif

    #if (defined CLD_ILLUMINANCE_MEASUREMENT) && (defined 
ILLUMINANCE_MEASUREMENT_CLIENT)
        /* Illuminance Measurement Cluster - Client */
        tsCLD_IlluminanceMeasurement sIlluminanceMeasurementClientCluster;
    #endif

    #if (defined CLD_ILLUMINANCE_LEVEL_SENSING) && (defined 
ILLUMINANCE_LEVEL_SENSING_CLIENT)
        tsCLD_IlluminanceLevelSensing sIlluminanceLevelSensingClientCluster;
    #endif

    #if (defined CLD_OCCUPANCY_SENSING) && (defined OCCUPANCY_SENSING_CLIENT)
        /* Occupancy Sensing Cluster - Client */
        tsCLD_OccupancySensing sOccupancySensingClientCluster;
    #endif

    #if (defined CLD_ZLL_COMMISSION) && (defined ZLL_COMMISSION_CLIENT)
        tsCLD_ZllCommission sZllCommissionClientCluster;
        tsCLD_ZllCommissionCustomDataStructure 
                          sZllCommissionClientCustomDataStructure;
    #endif

} tsZLO_ControlBridgeDevice;
```

3.20.3 Registration Function

The following **eZLO_RegisterControlBridgeEndPoint()** function is the endpoint registration function for a Control Bridge device.

> **teZCL_Status eZLO_RegisterControlBridgeEndPoint(**
> **uint8** *u8EndPointIdentifier*,
> **tfpZCL_ZCLCallBackFunction** *cbCallBack*,
> **tsZLO_ControlBridgeDevice** **psDeviceInfo*);

Description

This function is used to register an endpoint which will support a Control Bridge device. The function must be called after **eZCL_Initialise()**.

The specified identifier for the endpoint is a number in the range 1 to 240 (endpoint 0 is reserved for ZigBee use). Application endpoints are normally numbered consecutively starting at 1. The specified number must be less than or equal to the value of ZLO_NUMBER_OF_ENDPOINTS defined in the **zcl_options.h** file, which represents the highest endpoint number used for applications.

As part of this function call, you must specify a user-defined callback function that will be invoked when an event occurs that is associated with the endpoint. This callback function is defined according to the typedef:

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

You must also provide a pointer to a `tsZLO_ControlBridgeDevice` structure (see Section 3.20.2) which will be used to store all variables relating to the Control Bridge device associated with the endpoint. The `sEndPoint` and `sClusterInstance` fields of this structure are set by this function and must not be directly written to by the application.

The function may be called multiple times if more than one endpoint is being used - for example, if more than one Control Bridge device is housed in the same hardware, sharing the same module.

Parameters

u8EndPointIdentifier	Endpoint that is to be associated with the registered structure and callback function
cbCallBack	Pointer to the function that will be used to indicate events to the application for this endpoint
psDeviceInfo	Pointer to the structure that will act as storage for all variables related to the device being registered on this endpoint (see Section 3.20.2). The `sEndPoint` and `sClusterInstance` fields are set by this register function for internal use and must not be written to by the application

Returns

E_ZCL_SUCCESS
E_ZCL_FAIL
E_ZCL_ERR_PARAMETER_NULL
E_ZCL_ERR_PARAMETER_RANGE
E_ZCL_ERR_EP_RANGE
E_ZCL_ERR_CLUSTER_0
E_ZCL_ERR_CALLBACK_NULL
E_ZCL_ERR_CLUSTER_NULL
E_ZCL_ERR_SECURITY_RANGE
E_ZCL_ERR_CLUSTER_ID_RANGE
E_ZCL_ERR_MANUFACTURER_SPECIFIC
E_ZCL_ERR_ATTRIBUTE_TYPE_UNSUPPORTED
E_ZCL_ERR_ATTRIBUTE_ID_ORDER
E_ZCL_ERR_ATTRIBUTES_ACCESS

The above codes are described in the *ZCL User Guide (JN-UG-3132)*.

3.21 On/Off Sensor

The On/Off Sensor device is typically used in sensor nodes that issue control commands, e.g. an infra-red occupancy sensor.

- The Device ID is 0x0850
- The header file for the device is **on_off_sensor.h**
- The clusters supported by the device are listed in Section 3.21.1
- The device structure, `tsZLO_OnOffSensorDevice`, is listed in Section 3.21.2
- The endpoint registration function for the device, **eZLO_RegisterOnOffSensorEndPoint()**, is detailed in Section 3.21.3

3.21.1 Supported Clusters

The clusters used by the On/Off Sensor device are listed in the table below.

Server (Input) Side	**Client (Output) Side**
Mandatory	
Basic	On/Off
Identify	Identify
Optional	
Touchlink Commissioning	Touchlink Commissioning
	Level Control
	Colour Control
	Groups
	Scenes
	OTA Upgrade

Table 22: Clusters for On/Off Sensor

The mandatory attributes within each cluster for this device type are indicated in the *ZigBee Lighting and Occupancy Device Specification (15-0014-01)*.

3.21.2 Device Structure

The following `tsZLO_OnOffSensorDevice` structure is the shared structure for a On/Off Sensor device:

```
typedef struct
{
    tsZCL_EndPointDefinition sEndPoint;

    /* Cluster instances */
    tsZLO_OnOffSensorDeviceClusterInstances sClusterInstance;

    #if (defined CLD_BASIC) && (defined BASIC_SERVER)
        /* Basic Cluster - Server */
        tsCLD_Basic sBasicServerCluster;
    #endif

    #if (defined CLD_IDENTIFY) && (defined IDENTIFY_SERVER)
        /* Identify Cluster - Server */
        tsCLD_Identify sIdentifyServerCluster;
        tsCLD_IdentifyCustomDataStructure
                       sIdentifyServerCustomDataStructure;
    #endif

    /* Recommended Optional Server Cluster */
    #if (defined CLD_ZLL_COMMISSION) && (defined ZLL_COMMISSION_SERVER)
        tsCLD_ZllCommission sZllCommissionServerCluster;
        tsCLD_ZllCommissionCustomDataStructure
                            sZllCommissionServerCustomDataStructure;
    #endif

    /*
     * Mandatory client clusters
     */
    #if (defined CLD_IDENTIFY) && (defined IDENTIFY_CLIENT)
        /* Identify Cluster - Client */
        tsCLD_Identify sIdentifyClientCluster;
        tsCLD_IdentifyCustomDataStructure
                       sIdentifyClientCustomDataStructure;
    #endif

    #if (defined CLD_ONOFF) && (defined ONOFF_CLIENT)
        /* On/Off Cluster - Client */
        tsCLD_OnOff sOnOffClientCluster;
    #endif

    /* Recommended Optional Client CLuster */
    #if (defined CLD_LEVEL_CONTROL) && (defined LEVEL_CONTROL_CLIENT)
        /* Level Control Cluster - Client */
        tsCLD_LevelControl sLevelControlClientCluster;
        tsCLD_LevelControlCustomDataStructure
sLevelControlClientCustomDataStructure;
    #endif

    #if (defined CLD_SCENES) && (defined SCENES_CLIENT)
        /* Scenes Cluster - Client */
        tsCLD_Scenes sScenesClientCluster;
        tsCLD_ScenesCustomDataStructure sScenesClientCustomDataStructure;
    #endif

    #if (defined CLD_GROUPS) && (defined GROUPS_CLIENT)
        /* Groups Cluster - Client */
        tsCLD_Groups sGroupsClientCluster;
        tsCLD_GroupsCustomDataStructure sGroupsClientCustomDataStructure;
    #endif

    #if (defined CLD_COLOUR_CONTROL) && (defined COLOUR_CONTROL_CLIENT)
        /* Colour Control Cluster - Client */
        tsCLD_ColourControl sColourControlClientCluster;
        tsCLD_ColourControlCustomDataStructure
                            sColourControlClientCustomDataStructure;
    #endif

    #if (defined CLD_OTA) && (defined OTA_CLIENT)
        /* OTA cluster - Client */
        tsCLD_AS_Ota sCLD_OTA;
        tsOTA_Common sCLD_OTA_CustomDataStruct;
    #endif

    #if (defined CLD_ZLL_COMMISSION) && (defined ZLL_COMMISSION_CLIENT)
        tsCLD_ZllCommission sZllCommissionClientCluster;
        tsCLD_ZllCommissionCustomDataStructure
                            sZllCommissionClientCustomDataStructure;
    #endif

} tsZLO_OnOffSensorDevice;
```

3.21.3 Registration Function

The following **eZLO_RegisterOnOffSensorEndPoint()** function is the endpoint registration function for an On/Off Sensor device.

teZCL_Status eZLO_RegisterOnOffSensorEndPoint(
uint8 *u8EndPointIdentifier*,
tfpZCL_ZCLCallBackFunction *cbCallBack*,
tsZLO_OnOffSensorDevice **psDeviceInfo*);

Description

This function is used to register an endpoint which will support an On/Off Sensor device. The function must be called after **eZCL_Initialise()**.

The specified identifier for the endpoint is a number in the range 1 to 240 (endpoint 0 is reserved for ZigBee use). Application endpoints are normally numbered consecutively starting at 1. The specified number must be less than or equal to the value of ZLO_NUMBER_OF_ENDPOINTS defined in the **zcl_options.h** file, which represents the highest endpoint number used for applications.

As part of this function call, you must specify a user-defined callback function that will be invoked when an event occurs that is associated with the endpoint. This callback function is defined according to the typedef:

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

You must also provide a pointer to a `tsZLO_OnOffSensorDevice` structure (see Section 3.21.2) which will be used to store all variables relating to the On/Off Sensor device associated with the endpoint. The `sEndPoint` and `sClusterInstance` fields of this structure are set by this function and must not be directly written to by the application.

The function may be called multiple times if more than one endpoint is being used - for example, if more than one On/Off Sensor device is housed in the same hardware, sharing the same module.

Parameters

u8EndPointIdentifier	Endpoint that is to be associated with the registered structure and callback function
cbCallBack	Pointer to the function that will be used to indicate events to the application for this endpoint
psDeviceInfo	Pointer to the structure that will act as storage for all variables related to the device being registered on this endpoint (see Section 3.21.2). The `sEndPoint` and `sClusterInstance` fields are set by this register function for internal use and must not be written to by the application

Returns

E_ZCL_SUCCESS
E_ZCL_FAIL
E_ZCL_ERR_PARAMETER_NULL
E_ZCL_ERR_PARAMETER_RANGE
E_ZCL_ERR_EP_RANGE
E_ZCL_ERR_CLUSTER_0
E_ZCL_ERR_CALLBACK_NULL
E_ZCL_ERR_CLUSTER_NULL
E_ZCL_ERR_SECURITY_RANGE
E_ZCL_ERR_CLUSTER_ID_RANGE
E_ZCL_ERR_MANUFACTURER_SPECIFIC
E_ZCL_ERR_ATTRIBUTE_TYPE_UNSUPPORTED
E_ZCL_ERR_ATTRIBUTE_ID_ORDER
E_ZCL_ERR_ATTRIBUTES_ACCESS

The above codes are described in the *ZCL User Guide (JN-UG-3132)*.

Revision History

Version	Date	Comments
1.0	12 June 2018	First release
2.0	18 November 2019	Updated for K32W\JN5189

s

Important Notice

NXP Semiconductors

For online support resources and contact details of your local NXP office or distributor, refer to:

www.nxp.com