



# ZigBee Light Link User Guide

JN-UG-3091  
Revision 1.3  
3 August 2016



**ZigBee Light Link  
User Guide**

# Contents

<b>Preface</b>	<b>9</b>
Organisation	9
Conventions	10
Acronyms and Abbreviations	10
Related Documents	10
Support Resources	11
Trademarks	11
Chip Compatibility	11

## Part I: Concept and Development Information

<b>1. Introduction to ZigBee Light Link (ZLL)</b>	<b>15</b>
1.1 ZLL Objectives	15
1.2 ZLL Functionality	16
1.3 Wireless Networking	18
1.4 Touchlink Installation	18
1.5 Energy Saving	19
1.6 Interoperability and Certification	19
1.7 Software Architecture	20
1.8 Network Addresses	20
1.9 Security	21
1.10 Internet Connectivity	21
<b>2. ZLL Devices</b>	<b>23</b>
2.1 Clusters	23
2.2 Lighting Devices	24
2.2.1 On/Off Light	25
2.2.2 On/Off Plug-in Unit	25
2.2.3 Dimmable Light	26
2.2.4 Dimmable Plug-in Unit	26
2.2.5 Colour Light	27
2.2.6 Extended Colour Light	27
2.2.7 Colour Temperature Light	28
2.3 Controller Devices	29
2.3.1 Colour Controller	29
2.3.2 Colour Scene Controller	30
2.3.3 Non-Colour Controller	30

## Contents

2.3.4 Non-Colour Scene Controller	31
2.3.5 Control Bridge	31
2.3.6 On/Off Sensor	32
<b>3. ZLL Application Development</b>	<b>33</b>
3.1 Development Resources and Installation	33
3.2 ZLL Programming Resources	34
3.2.1 Core Resources	34
3.2.2 Cluster-specific Resources	34
3.3 Function Prefixes	35
3.4 Development Phases	35
3.5 Building an Application	36
3.5.1 Compile-Time Options	36
3.5.2 ZigBee Network Parameters	37
3.5.3 Building and Loading the Application Binary	37
<b>4. ZLL Application Coding</b>	<b>39</b>
4.1 ZLL Programming Concepts	39
4.1.1 Shared Device Structures	39
4.1.2 Addressing	41
4.1.3 OS Resources	41
4.2 Initialisation	42
4.3 Callback Functions	43
4.4 Network Formation/Joining	43
4.5 Reading Attributes	44
4.6 Writing Attributes	46
4.7 Handling Stack and Timer Events	49
4.8 Servicing Timing Requirements	50
<b>Part II: ZLL Clusters</b>	
<b>5. ZCL Clusters</b>	<b>53</b>
5.1 Basic Cluster	54
5.1.1 Mandatory Attributes for ZLL	54
5.1.2 Compile-Time Options	54
5.2 Identify Cluster	55
5.2.1 Mandatory Attribute for ZLL	55
5.2.2 Enhanced Functionality for ZLL	55
5.2.3 Compile-Time Options	55

5.3	Groups Cluster	56
5.3.1	Mandatory Attribute for ZLL	56
5.3.2	Compile-Time Options	56
5.4	Scenes Cluster	57
5.4.1	Mandatory Attributes for ZLL	57
5.4.2	Enhanced Functionality for ZLL	57
5.4.3	Compile-Time Options	57
5.5	On/Off Cluster	59
5.5.1	Mandatory Attributes for ZLL	59
5.5.2	Enhanced Functionality for ZLL	59
5.5.3	Compile-Time Options	59
5.6	Level Control Cluster	61
5.6.1	Mandatory Attributes for ZLL	61
5.6.2	Compile-Time Options	61
5.7	Colour Control Cluster	62
5.7.1	Mandatory Attributes for ZLL	62
5.7.2	Enhanced Functionality for ZLL	63
5.7.3	Compile-Time Options	64
<b>6.</b>	<b>ZLL Commissioning Cluster</b>	<b>67</b>
6.1	Overview	67
6.2	ZLL Commissioning Cluster Structure and Attributes	68
6.3	Commissioning Operations	68
6.4	Using Touchlink	69
6.4.1	Creating a ZLL Network	70
6.4.2	Adding to an Existing Network	72
6.4.3	Updating Network Settings	73
6.4.4	Stealing a Node	74
6.5	Using the Commissioning Utility	75
6.6	ZLL Commissioning Events (Touchlink)	77
6.6.1	Touchlink Command Events	79
6.6.2	Commissioning Utility Command Events	80
6.7	Functions	80
6.7.1	Touchlink Functions	81
	eZLL_RegisterCommissionEndPoint	82
	eCLD_ZllCommissionCreateCommission	83
	eCLD_ZllCommissionCommandScanReqCommandSend	84
	eCLD_ZllCommissionCommandScanRspCommandSend	85
	eCLD_ZllCommissionCommandDeviceInfoReqCommandSend	86
	eCLD_ZllCommissionCommandDeviceInfoRspCommandSend	87
	eCLD_ZllCommissionCommandDeviceIdentifyReqCommandSend	88
	eCLD_ZllCommissionCommandFactoryResetReqCommandSend	89
	eCLD_ZllCommissionCommandNetworkStartReqCommandSend	90

## Contents

eCLD_ZIICommissionCommandNetworkStartRspCommandSend	91
eCLD_ZIICommissionCommandNetworkJoinRouterReqCommandSend	92
eCLD_ZIICommissionCommandNetworkJoinRouterRspCommandSend	93
eCLD_ZIICommissionCommandNetworkJoinEndDeviceReqCommandSend	94
eCLD_ZIICommissionCommandNetworkJoinEndDeviceRspCommandSend	95
eCLD_ZIICommissionCommandNetworkUpdateReqCommandSend	96
6.7.2 Commissioning Utility Functions	97
eCLD_ZIIUtilityCreateUtility	98
eCLD_ZIIUtilityCommandEndpointInformationCommandSend	99
eCLD_ZIIUtilityCommandGetGroupIdReqCommandSend	100
eCLD_ZIIUtilityCommandGetGroupIdRspCommandSend	101
eCLD_ZIIUtilityCommandGetEndpointListReqCommandSend	102
eCLD_ZIIUtilityCommandGetEndpointListRspCommandSend	103
eCLD_ZIIUtilityCommandHandler	104
<b>6.8 Structures</b>	<b>105</b>
6.8.1 tsZLL_CommissionEndpoint	105
6.8.2 tsZLL_CommissionEndpointClusterInstances	106
6.8.3 tsCLD_ZIICommissionCustomDataStructure	106
6.8.4 tsCLD_ZIICommissionCallBackMessage	107
6.8.5 tsCLD_ZIICommission_ScanReqCommandPayload	108
6.8.6 tsCLD_ZIICommission_ScanRspCommandPayload	108
6.8.7 tsCLD_ZIICommission_DeviceInfoReqCommandPayload	110
6.8.8 tsCLD_ZIICommission_DeviceInfoRspCommandPayload	111
6.8.9 tsCLD_ZIICommission_IdentifyReqCommandPayload	111
6.8.10 tsCLD_ZIICommission_FactoryResetReqCommandPayload	112
6.8.11 tsCLD_ZIICommission_NetworkStartReqCommandPayload	112
6.8.12 tsCLD_ZIICommission_NetworkStartRspCommandPayload	114
6.8.13 tsCLD_ZIICommission_NetworkJoinRouterReqCommandPayload	115
6.8.14 tsCLD_ZIICommission_NetworkJoinRouterRspCommandPayload	116
6.8.15 tsCLD_ZIICommission_NetworkJoinEndDeviceReqCommandPayload	117
6.8.16 tsCLD_ZIICommission_NetworkJoinEndDeviceRspCommandPayload	118
6.8.17 tsCLD_ZIICommission_NetworkUpdateReqCommandPayload	119
6.8.18 tsCLD_ZIIUtilityCustomDataStructure	119
6.8.19 tsCLD_ZIIUtilityCallBackMessage	120
6.8.20 tsCLD_ZIIUtility_EndpointInformationCommandPayload	121
<b>6.9 Enumerations</b>	<b>122</b>
6.9.1 Touchlink Event Enumerations	122
6.9.2 Commissioning Utility Event Enumerations	122
<b>6.10 Compile-Time Options</b>	<b>123</b>

## Part III: General Reference Information

<b>7. ZLL Core Functions</b>	<b>127</b>
eZLL_Initialise	128
eZLL_Update100mS	129
eZLL_RegisterOnOffLightEndPoint	130
eZLL_RegisterOnOffPlugEndPoint	132
eZLL_RegisterDimmableLightEndPoint	134
eZLL_RegisterDimmablePlugEndPoint	136
eZLL_RegisterColourLightEndPoint	138
eZLL_RegisterExtendedColourLightEndPoint	140
eZLL_RegisterColourTemperatureLightEndPoint	142
eZLL_RegisterColourRemoteEndPoint	144
eZLL_RegisterColourSceneRemoteEndPoint	146
eZLL_RegisterNonColourRemoteEndPoint	148
eZLL_RegisterNonColourSceneRemoteEndPoint	150
eZLL_RegisterControlBridgeEndPoint	152
eZLL_RegisterOnOffSensorEndPoint	154
<b>8. ZLL Structures</b>	<b>157</b>
8.1 Device Structures	157
8.1.1 tsZLL_OnOffLightDevice	157
8.1.2 tsZLL_OnOffPlugDevice	158
8.1.3 tsZLL_DimmableLightDevice	160
8.1.4 tsZLL_DimmablePlugDevice	161
8.1.5 tsZLL_ColourLightDevice	162
8.1.6 tsZLL_ExtendedColourLightDevice	163
8.1.7 tsZLL_ColourTemperatureLightDevice	166
8.1.8 tsZLL_ColourRemoteDevice	167
8.1.9 tsZLL_ColourSceneRemoteDevice	169
8.1.10 tsZLL_NonColourRemoteDevice	170
8.1.11 tsZLL_NonColourSceneRemoteDevice	172
8.1.12 tsZLL_ControlBridgeDevice	173
8.1.13 tsZLL_OnOffSensorDevice	175
8.2 Other Structures	178
8.2.1 tsCLD_ZllDeviceRecord	178

## **Contents**

---

## Preface

This manual provides an introduction to the ZigBee Light Link (ZLL) application profile and describes the use of the NXP ZigBee ZLL Application Programming Interface (API) for the JN516x wireless microcontrollers. The manual contains both operational and reference information relating to the ZLL API, including descriptions of the C functions and associated resources (e.g. structures and enumerations).

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

- Information on ZigBee PRO wireless networks is provided in the *ZigBee PRO Stack User Guide (JN-UG-3101)*, available from NXP.
- The ZLL profile is defined in the *ZigBee Light Link Profile Specification (11-0037-10)*, available from the ZigBee Alliance at [www.zigbee.org](http://www.zigbee.org).



**Important:** Clusters that are part of the ZigBee Cluster Library (ZCL) but used by the ZLL profile are detailed in the *ZCL User Guide (JN-UG-3103)* from NXP, which you should use in conjunction with this manual.

---

## Organisation

This manual is divided into three parts:

- **Part I: Concept and Development Information** comprises four chapters:
  - **Chapter 1** introduces the principles of ZigBee Light Link (ZLL)
  - **Chapter 2** describes the devices available in the ZLL application profile
  - **Chapter 3** provides an overview of ZLL application development
  - **Chapter 4** describes the essential aspects of coding a ZLL application using NXP's ZLL Application Programming Interface (API)
- **Part II: ZLL Clusters** comprises two chapters:
  - **Chapter 5** describes the ZLL-specific implementation of clusters from the ZigBee Cluster Library (ZCL)
  - **Chapter 6** details the ZLL Commissioning cluster, including API resources
- **Part III: General Reference Information** comprises two chapters:
  - **Chapter 7** details the ZLL core functions, including initialisation and device registration functions
  - **Chapter 8** details general structures of the ZLL API

---

## Conventions

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

Function parameters are represented in *italics* type.

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



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



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



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

---

## Acronyms and Abbreviations

API	Application Programming Interface
SDK	Software Developer's Kit
ZCL	ZigBee Cluster Library
ZLL	ZigBee Light Link

---

## Related Documents

JN-UG-3101	ZigBee PRO Stack User Guide
JN-UG-3103	ZigBee Cluster Library User Guide
JN-UG-3075	JenOS User Guide
JN-UG-3098	Beyond Studio for NXP Installation and User Guide
JN-AN-1171	ZigBee Light Link Solution Application Note
JN-AN-1194	ZigBee Gateway Application Note
11-0037-10	ZigBee Light Link Profile Specification [from ZigBee Alliance]
075123	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](http://www.nxp.com/products/interface-and-connectivity/wireless-connectivity)

ZigBee resources can be accessed from the ZigBee page, which can be reached via the short-cut [www.nxp.com/zigbee](http://www.nxp.com/zigbee).

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

---

## Trademarks

All trademarks are the property of their respective owners.

---

## Chip Compatibility

The ZLL software described in this manual can be used on the NXP JN516x family of wireless microcontrollers with the exception of the JN5161 device. However, the supported devices will be referred to as JN516x.

## ***Preface***

# Part I: Concept and Development Information



---

# 1. Introduction to ZigBee Light Link (ZLL)

The ZigBee Alliance has developed the ZigBee Light Link (ZLL) application profile for ZigBee PRO in order to meet the needs of lighting solutions in the consumer market. This chapter introduces the purpose and concepts of ZigBee Light Link before the NXP implementation is detailed in the rest of this manual.

The ZLL Application Profile ID is 0xC05E and the profile is defined in the *ZigBee Light Link Profile Specification (11-0037-10)*, available from the ZigBee Alliance. However, this User Guide together with the *ZigBee Cluster Library User Guide (JN-UG-3103)* should provide all the necessary information to use the NXP implementation of the profile.



**Note:** ZigBee Light Link operates in conjunction with the ZigBee PRO wireless network protocol. If you are not already familiar with ZigBee PRO, you are advised to read at least the first two chapters of the *ZigBee PRO Stack User Guide (JN-UG-3101)*.

---

## 1.1 ZLL Objectives

ZigBee Light Link brings wireless technology to lighting solutions for the home, opening up a new world of lighting to consumers in a user-friendly and accessible way.

The objectives of ZigBee Light Link can be summarised as follows:

- Target the DIY consumer market as well as small professional installations
- Provide an easy and intuitive installation experience
- Provide the consumer with new, rich lighting functionality, including remote control, programmable timer control and mood lighting
- Allow energy saving (and power cost savings) through occupancy sensing and energy monitoring
- Provide a framework for interoperability between products from different manufacturers

The operational principles for attaining the above objectives are described in the remainder of this chapter.

## 1.2 ZLL Functionality

A ZigBee Light Link system is a wireless network which contains two general categories of nodes - those that are used to send control commands, and those that receive and execute control commands:

- **Controller nodes** - these may include:
  - Light switches (e.g. on walls)
  - Occupancy sensors
  - Remote control unit(s)
  - Smart phones
  - Computing devices (e.g. PC or tablet)
- **Light (controlled) nodes** - these may include:
  - Monochrome lamps (in ceiling lights, wall lights, table lamps, etc)
  - Colour lamps

Thus, one or more lamps may be controlled (switched on, switched off, dimmed, colour adjusted) from a controlling node in the system - for example, the user may choose to dim a table lamp next to the TV from a remote control unit while sitting on the sofa. An example ZLL system is illustrated in Table 1 on page 17.

The functionality of a ZLL system goes way beyond remotely switching lamps on and off, or dimming lamps:

- **Colour lamps:** Advances in LED technology have resulted in the proliferation of LED-based lights, including colour lamps with configurable colour (as well as overall brightness). ZLL can be used to remotely control this colour setting.
- **Mood lighting:** ZLL allows lights to be adjusted to create a certain 'mood' or ambiance, as follows:
  - Lamps can be grouped, where a particular group may be selected to create a certain mood (e.g. watching TV from sofa, reading in favourite armchair or eating at dining table). A system can support a number of a groups and an individual lamp may belong to more than one group.
  - In connection with groups, the brightness of the individual lamps may be adjusted to create a 'scene' corresponding to a certain mood (e.g. table lamp next to TV at low brightness, table lamp behind sofa at medium brightness and spot lamp next to armchair at full brightness). A system can support a number of scenes.
  - Colour settings can be adjusted, as described for colour lamps above.
- **Programmable timers:** The system can be programmed to automatically adjust lights at certain times (e.g. to switch an outside light on and off).

ZigBee Light Link allows a home lighting system (such as the one in Figure 1) to be connected to and controlled from the Internet (see Section 1.10). This enables the system to be controlled from a smart phone, tablet or PC located inside or outside the house, via an IP router (gateway).

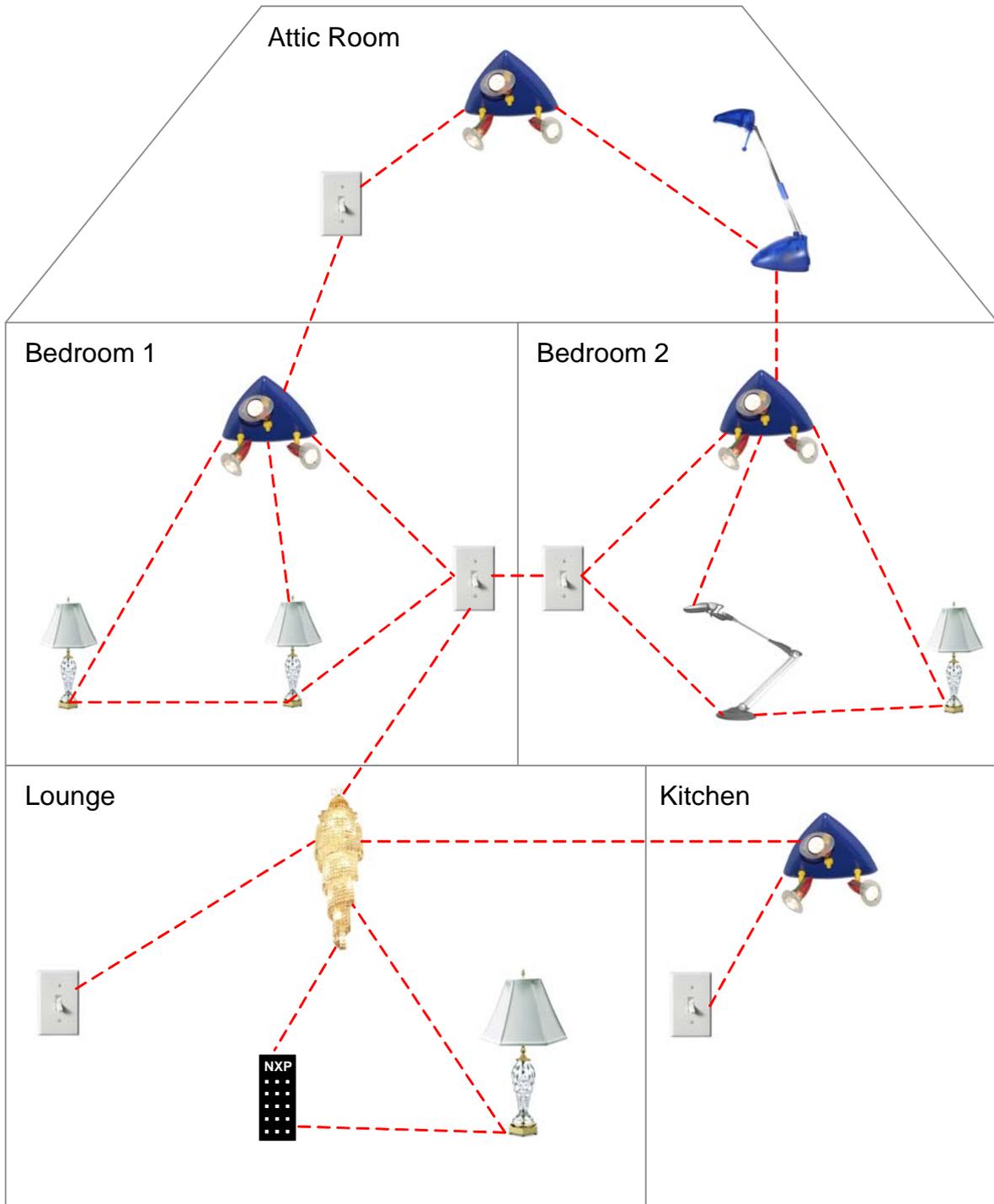


Figure 1: Example ZLL Network

---

## 1.3 Wireless Networking

ZigBee Light Link is a public application profile that has been devised by the ZigBee Alliance to support consumer lighting solutions based on the ZigBee PRO wireless network protocol. However, unlike a conventional ZigBee PRO network, a ZLL system does not have a ZigBee Co-ordinator. Instead, network formation/joining is performed using a special commissioning application, which can be used on any node (usually a remote control unit).

A Mesh network topology is employed. Therefore, for maximum routing flexibility, all the network nodes of a ZLL system should be ZigBee Routers (although ZigBee End Devices are permitted, they cannot perform Mesh routing).

The manufacturer application that runs on a ZLL node provides the interface between the ZLL profile software and the hardware of the node (e.g. the physical switch mechanism of a lamp).



**Note:** The software architecture for ZLL, in terms of a protocol stack, is described in more detail in [Section 1.7](#).

The ZLL profile contains a number of 'devices', which are ZigBee software entities used to implement particular functionality on a node - for example, the 'On/Off Light' device is used to switch a lamp on and off. The set of devices used in a node determines the total functionality of the node.

Each ZLL device uses a number of clusters, where most clusters used in the ZLL profile come from the ZigBee Cluster Library (ZCL). Complete lists of the devices and associated clusters used by the ZLL profile are provided in [Chapter 2](#).

---

## 1.4 Touchlink Installation

A ZLL system is a ZigBee PRO wireless network but benefits from a simplified installation method in order to appeal to the consumer market. This method is known as Touchlink and minimises user participation, allowing off-the-shelf products to be quickly and easily installed by the householder.

Touchlink removes the need for a ZigBee Co-ordinator in the network formation and join processes. The method uses a special commissioning application (based on the ZLL Commissioning cluster) which is run on the nodes. The node that initiates the network formation/join operation is known as the 'initiator' - this node will often be a remote control unit but could be another node, such as a lamp. Touchlink simply requires the initiator node to be brought close to the node to be included in the network and the commissioning to be started (e.g. by pressing a button).

Commissioning involves three sets of command exchanges between the nodes:

1. **Discovery:** The initiator node performs a scan for ZLL nodes in its vicinity, based on received signal strength. This results in a list of detected nodes which, for each node, includes information on network capabilities, device type and whether the node is using its factory settings. If more than one node is found, the application on the initiator must decide which node(s) to commission.
2. **Transfer of network settings:** The initiator then requests and receives the network settings of the node(s) of interest.
3. **Request network formation or join:** The initiator then requests a node of interest to either form a new network or join an existing network.

Touchlink employs inter-PAN communication for commissioning messages.

Security settings may also be established during commissioning - see [Section 1.9](#).

---

## 1.5 Energy Saving

A ZLL system can result in energy saving and associated cost savings for a household. The following may be employed to achieve this:

- **Scenes and timers:** Energy savings can be achieved through the careful configuration of 'scenes' and timers (see [Section 1.2](#)), to ensure that no more energy is consumed than is actually needed.
- **Occupancy sensors:** Infra-red or movement sensors can be used to switch on lights only when a person is detected (and switch off the lights when a person is no longer detected). This method may be very useful for controlling lights in a corridor or garage, or outside lights.
- **Energy monitoring:** When used in conjunction with the ZigBee Home Automation (HA) profile, the power consumption of a ZLL system may be monitored.

---

## 1.6 Interoperability and Certification

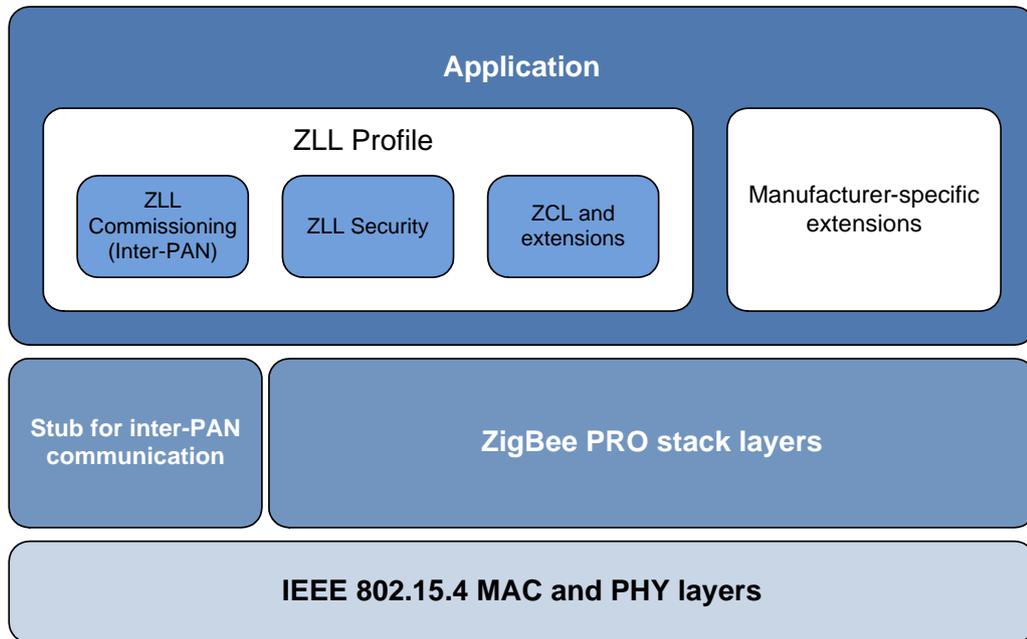
ZigBee Light Link provides a framework of interoperability between products from different manufacturers. This is formalised through a ZLL certification and compliance programme, in which completed products are tested for compliance to the ZLL profile and, if successful, are ZLL certified.

Thus, a product developed and certified to the ZLL profile will be able operate with other certified products in a ZLL system, irrespective of their manufacturers. This is an important feature for the consumer market targeted by ZLL.

In addition, the ZLL profile is designed to be interoperable at the network layer with other public ZigBee application profiles, particularly Home Automation (HA).

## 1.7 Software Architecture

ZigBee Light Link operates in conjunction with the ZigBee PRO wireless network protocol. The software stack which runs on each ZLL node is illustrated in [Figure 2](#).



**Figure 2: ZLL Software Stack**

The main features of the above stack are as follows:

- The (manufacturer) application uses the ZLL profile, interfaces to the underlying ZigBee PRO stack layers and controls the lighting hardware of the node. The ZLL profile includes:
  - ZLL commissioning (including Touchlink) - see [Section 1.4](#)
  - ZLL security - see [Section 1.9](#)
  - ZLL resources (ZCL clusters and extensions)

Manufacturer-specific extensions can also be used to supplement ZLL.

- The normal ZigBee PRO stack layers are supplemented by a stub to support inter-PAN communications. The ZigBee PRO stack layers are described in the *ZigBee PRO Stack User Guide (JN-UG-3101)*.

## 1.8 Network Addresses

ZLL networks use 16-bit network (short) addresses to identify nodes. The assignment of network addresses to nodes in a ZLL network is not performed in the same way as in a classic ZigBee PRO network, in that this assignment is not random. Only a ZLL controller device (see [Section 1.8](#)) is able to assign network addresses, from an allocated range of possible addresses. If another controller device is added to the

network, it will inherit a portion of this address range for its own allocation, where this portion is specified in the join request for the new node. For more information on network address assignment in ZLL networks, refer to the section “*Network address assignment*” in the *ZLL Specification*.

---

## 1.9 Security

ZigBee Light Link cannot use ZigBee security in its standard form, since there is no Co-ordinator or Trust Centre in a ZLL system. ZLL uses a network-level security in which the same network key is used by all nodes in the network to encrypt/decrypt communications between them.

The network key is generated randomly by the initiator node when the network is formed (see [Section 1.4](#)) and is unique to the network. The distribution of this network key to nodes subsequently joining the network is secured using the ZLL master key which is pre-installed in all ZLL-certified nodes during manufacture.

The security set-up process during the commissioning of a node (after the node has been detected and its network settings obtained) is as follows:

1. If the target node is to be the first node of the network (network formation), the initiator node generates a random network key and encrypts it using the ZLL master key (and stores the encrypted network key locally).
2. The initiator node sends the encrypted network key to the target node as part of the request to form or join the network.
3. The target node decrypts the received network key using the ZLL master key (and stores the network key locally).
4. All future communications from/to this node will be encrypted with the network key.

---

## 1.10 Internet Connectivity

ZigBee Light Link offers the possibility of controlling the lights in a ZLL system via the Internet. Thus, this control can be performed from any Internet-connected device (PC, tablet, smart phone) located anywhere in the World (e.g. while on holiday in another country).

Access from the Internet requires the ZLL system to include an IP router or gateway (connected to the Internet) as one of the network nodes. A gateway solution is described in the Application Note *ZigBee Gateway (JN-AN-1194)*, available from NXP.

In addition to the real-time control of a ZLL system over the Internet, the system could also be configured from a device on the Internet (e.g. groups, scenes and timers).

**Chapter 1**  
***Introduction to ZigBee Light Link (ZLL)***

## 2. ZLL Devices

This chapter details the ZigBee devices available in the ZigBee Light Link profile and the clusters that they use. The ZLL devices are divided into two categories:

- **Lighting devices:** These are used in ZLL light nodes, such as a lamp, and are described in [Section 2.2](#).
- **Controller devices:** These are used in ZLL controller nodes, such as a remote control unit, and are described in [Section 2.3](#).

In the sections referenced above, the server clusters and client clusters used on each ZLL device are listed. First, all the clusters used in the ZLL devices are introduced in [Section 2.1](#).



**Note:** All ZLL devices use the Basic cluster (from the ZCL) as a server cluster. This cluster is detailed in the *ZCL User Guide (JN-UG-3103)*.

### 2.1 Clusters

The ZLL profile uses certain clusters from the ZigBee Cluster Library (ZCL) and also defines a cluster of its own. All the clusters used by the ZLL profile are listed in [Table 1](#) and outlined below. In the table, the clusters from the ZCL and the one defined by the ZLL profile are listed separately.

Category	Cluster	Cluster ID
ZCL	Basic	0x0000
	Identify	0x0003
	Groups	0x0004
	Scenes	0x0005
	On/Off	0x0006
	Level Control	0x0008
	Colour Control	0x0300
ZLL	ZLL Commissioning	0x1000

**Table 1: Clusters used by ZLL**



**Note :** The ZLL Commissioning cluster is detailed in this manual ([Chapter 6](#)). Only essential information on the ZCL clusters is given in this manual ([Chapter 5](#)) - they are fully detailed in the *ZCL User Guide (JN-UG-3103)*.

## 2.2 Lighting Devices

This section details the clusters used by the ZLL lighting devices. These software devices are included in the physical ZLL nodes that are controlled, e.g. lamps.

The ZLL lighting devices and their Device IDs are summarised in [Table 2](#) below. The table also indicates whether each device can support either one or both of the ZLL Commissioning cluster server and ZLL Commissioning cluster client.

ZLL Device	Device ID	ZLL Commissioning Cluster			Reference
		Server	Server/Client	Client	
On/Off Light	0x0000	✓	✓	✗	<a href="#">Section 2.2.1</a>
On/Off Plug-in Unit	0x0010	✓	✓	✗	<a href="#">Section 2.2.2</a>
Dimmable Light	0x0100	✓	✓	✗	<a href="#">Section 2.2.3</a>
Dimmable Plug-in Unit	0x0110	✓	✓	✗	<a href="#">Section 2.2.4</a>
Colour Light	0x0200	✓	✓	✗	<a href="#">Section 2.2.5</a>
Extended Colour Light	0x0210	✓	✓	✗	<a href="#">Section 2.2.6</a>
Colour Temperature Light	0x0220	✓	✓	✗	<a href="#">Section 2.2.7</a>

**Table 2: ZLL Lighting Devices**



**Note 1:** All clusters used by the ZLL lighting devices are server-side clusters.

**Note 2:** None of the ZLL lighting devices needs to support the 'utility' functionality of the ZLL Commissioning cluster.

## 2.2.1 On/Off Light

The On/Off Light device is typically used in nodes that contain a lamp which can simply be switched on and off.

Its Device ID is 0x0000.

The clusters supported by this device are as follows:

Server (Input) Side Clusters	Client (Output) Side Clusters
Basic	
Identify	
Groups	
Scenes	
On/Off	
Level Control *	

**Table 3: Clusters for On/Off Light Device**

\* Level Control cluster is supported so that 'change level' commands (for example, issued by a Non-Colour Controller device) can control an On/Off Light. If the light is off, increasing the level will switch on the light. If the light is on, decreasing the level to the minimum value will switch off the light. This provides a good user experience when controlling a mixed group of Dimmable Lights and On/Off Lights.

## 2.2.2 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.

Its Device ID is 0x0010.

The clusters supported by this device are as follows:

Server (Input) Side Clusters	Client (Output) Side Clusters
Basic	
Identify	
Groups	
Scenes	
On/Off	
Level Control *	

**Table 4: Clusters for On/Off Plug-in Unit Device**

\* Level Control cluster is supported for similar reasons to those described for the On/Off Light device in [Section 2.2.1](#).

---

### 2.2.3 Dimmable Light

The Dimmable Light device is typically used in nodes that contain a lamp with adjustable brightness.

Its Device ID is 0x0100.

The clusters supported by this device are as follows:

Server (Input) Side Clusters	Client (Output) Side Clusters
Basic	
Identify	
Groups	
Scenes	
On/Off	
Level Control	

**Table 5: Clusters for Dimmable Light Device**

---

### 2.2.4 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).

Its Device ID is 0x0110.

The clusters supported by this device are as follows:

Server (Input) Side Clusters	Client (Output) Side Clusters
Basic	
Identify	
Groups	
Scenes	
On/Off	
Level Control	

**Table 6: Clusters for Dimmable Plug-in Unit Device**

## 2.2.5 Colour Light

The 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 loop and XY.

Its Device ID is 0x0200.

The clusters supported by this device are as follows:

Server (Input) Side Clusters	Client (Output) Side Clusters
Basic	
Identify	
Groups	
Scenes	
On/Off	
Level Control	
Colour Control	

**Table 7: Clusters for Colour Light Device**

## 2.2.6 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 colour temperature, in addition to the colour parameters supported by the Colour Light device (see [Section 2.2.5](#)).

Its Device ID is 0x0210.

The clusters supported by this device are as follows:

Server (Input) Side Clusters	Client (Output) Side Clusters
Basic	
Identify	
Groups	
Scenes	
On/Off	
Level Control	
Colour Control	

**Table 8: Clusters for Extended Colour Light Device**

---

## 2.2.7 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.

Its Device ID is 0x0220.

The clusters supported by this device are as follows:

Server (Input) Side Clusters	Client (Output) Side Clusters
Basic	
Identify	
Groups	
Scenes	
On/Off	
Level Control	
Colour Control	

**Table 9: Clusters for Colour Temperature Light Device**

## 2.3 Controller Devices

This section details the clusters used by the ZLL controller devices. These software devices are included in the physical ZLL nodes that issue control commands, e.g. a remote control unit.

The ZLL controller devices and their Device IDs are summarised in [Table 10](#) below. The table also indicates whether each device can support either one or both of the ZLL Commissioning cluster server and ZLL Commissioning cluster client.

ZLL Device	Device ID	ZLL Commissioning Cluster			Reference
		Server	Server/Client	Client	
Colour Controller	0x0800	✓	✓	✓	<a href="#">Section 2.3.1</a>
Colour Scene Controller	0x0810	✓	✓	✓	<a href="#">Section 2.3.2</a>
Non-Colour Controller	0x0820	✓	✓	✓	<a href="#">Section 2.3.3</a>
Non-Colour Scene Controller	0x0830	✓	✓	✓	<a href="#">Section 2.3.4</a>
Control Bridge	0x0840	✓	✓	✓	<a href="#">Section 2.3.5</a>
On/Off Sensor	0x0850	✓	✓	✓	<a href="#">Section 2.3.6</a>

**Table 10: ZLL Controller Devices**

### 2.3.1 Colour Controller

The Colour Controller device is typically used in nodes that issue ZLL colour-control commands, e.g. a remote control unit for colour lamps.

Its Device ID is 0x0800.

The clusters supported by this device are as follows:

Server (Input) Side Clusters	Client (Output) Side Clusters
Basic	
	Identify
	Groups
	On/Off
	Level Control
	Colour Control
ZLL Commissioning	ZLL Commissioning

**Table 11: Clusters for Colour Controller Device**

---

## 2.3.2 Colour Scene Controller

The Colour Scene Controller device is typically used in nodes that issue ZLL colour-control commands and that support ‘scenes’, e.g. a remote control unit for colour lamps.

Its Device ID is 0x0810.

The clusters supported by this device are as follows:

Server (Input) Side Clusters	Client (Output) Side Clusters
Basic	
	Identify
	Groups
	Scenes
	On/Off
	Level Control
	Colour Control
ZLL Commissioning	ZLL Commissioning

**Table 12: Clusters for Colour Scene Controller Device**

---

## 2.3.3 Non-Colour Controller

The Non-Colour Controller device is typically used in nodes that issue ZLL control commands that are not related to colour, e.g. a remote control unit for monochrome lamps.

Its Device ID is 0x0820.

The clusters supported by this device are as follows:

Server (Input) Side Clusters	Client (Output) Side Clusters
Basic	
	Identify
	Groups
	On/Off
	Level Control
ZLL Commissioning	ZLL Commissioning

**Table 13: Clusters for Non-Colour Controller Device**

## 2.3.4 Non-Colour Scene Controller

The Non-Colour Scene Controller device is typically used in nodes that issue ZLL control commands that are not related to colour and that support 'scenes', e.g. a remote control unit for monochrome lamps.

Its Device ID is 0x0830.

The clusters supported by this device are as follows:

Server (Input) Side Clusters	Client (Output) Side Clusters
Basic	
	Identify
	Groups
	Scenes
	On/Off
	Level Control
ZLL Commissioning	ZLL Commissioning

**Table 14: Clusters for Non-Colour Scene Controller Device**

## 2.3.5 Control Bridge

The Control Bridge device is typically used in nodes that relay ZLL control commands issued from another network, e.g. an Internet router with a ZLL interface.

Its Device ID is 0x0840.

The clusters supported by this device are as follows:

Server (Input) Side Clusters	Client (Output) Side Clusters
Basic	
	Identify
	Groups
	Scenes
	On/Off
	Level Control
	Colour Control
ZLL Commissioning	ZLL Commissioning

**Table 15: Clusters for Control Bridge Device**

---

## 2.3.6 On/Off Sensor

The On/Off Sensor device is typically used in sensor nodes that issue ZLL control commands, e.g. an infra-red occupancy sensor.

Its Device ID is 0x0850.

The clusters supported by this device are as follows:

Server (Input) Side Clusters	Client (Output) Side Clusters
Basic	
	Identify
	Groups
	Scenes
	On/Off
	Level Control
	Colour Control
ZLL Commissioning	ZLL Commissioning

**Table 16: Clusters for On/Off Sensor Device**

---

## 3. ZLL Application Development

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

- Development resources and their installation ([Section 3.1](#))
- ZLL programming resources ([Section 3.2](#))
- API functions ([Section 3.3](#))
- Development phases ([Section 3.4](#))
- Building an application ([Section 3.5](#))

Application coding is described separately in [Chapter 4](#).

---

### 3.1 Development Resources and Installation

NXP provide a wide range of resources to aid in the development of ZigBee Light Link applications for the JN516x wireless microcontrollers (see [“Chip Compatibility” on page 11](#)). A ZLL application is developed as a ZigBee PRO application that uses the NXP ZigBee PRO APIs in conjunction with JenOS (Jennic Operating System), together with ZLL-specific and ZCL resources. All resources are available from the Wireless Connectivity area of the NXP web site (see [“Support Resources” on page 11](#)) and are outlined below.

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

- **HA/ZLL SDK (JN-SW-4168):** This installer is shared with Home Automation (HA), and contains the ZigBee PRO stack and the ZLL profile software, including a number of C APIs:
  - ZLL and ZCL APIs
  - ZigBee PRO APIs
  - JenOS APIs
  - JN516x Integrated Peripherals API

In addition, the ZPS and JenOS Configuration Editors are provided in this installer (these are plug-ins for ‘BeyondStudio for NXP’ - see below).

- **BeyondStudio for NXP (JN-SW-4141):** This installer contains the toolchain that you will use in creating an application, including:
  - ‘Beyond Studio for NXP’ IDE (Integrated Development Environment)
  - Integrated JN51xx compiler
  - Integrated JN516x Flash Programmer

For full details of the toolchain and installation instructions for the toolchain and SDK, refer to the *BeyondStudio for NXP Installation and User Guide (JN-UG-3098)*.

A ZLL demonstration application is provided in the Application Note *ZigBee Light Link Solution (JN-AN-1171)*, available from the NXP web site.

## 3.2 ZLL Programming Resources

The NXP ZLL API contains a range of resources (functions, structures, etc), including:

- Core resources (e.g. for initialising the API and registering device endpoints)
- Cluster-specific resources

These resources are introduced in the sub-sections below.

---

### 3.2.1 Core Resources

The core resources of the ZLL profile handle the basic operations required in a ZLL network, irrespective of the clusters used. Some of these resources are provided in the ZLL API, and some are provided in the ZCL and ZigBee PRO APIs.

- Functions for the following operations are provided in the ZLL API and are detailed in [Chapter 7](#):
  - Initialising the ZLL API (one function)
  - Registering a device endpoint on a ZLL node (one function per device)
- Functions for the following operations are provided in the ZCL and are detailed in the *ZCL User Guide (JN-UG-3103)*:
  - Requesting a read access to cluster attributes on a remote device
  - Requesting a write access to cluster attributes on a remote device
  - Handling events on a ZLL device

Use of the above functions is described in [Chapter 4](#).

---

### 3.2.2 Cluster-specific Resources

A ZLL device uses certain mandatory and optional ZigBee clusters (for details, refer to [Chapter 2](#)). The clusters supported by the NXP ZLL software are listed below.

- Clusters from the ZCL are as follows (also refer to [Chapter 5](#)):
  - Basic
  - Identify
  - Groups
  - Scenes
  - On/Off
  - Level Control
  - Colour Control
- Clusters from the ZLL profile are:
  - ZLL Commissioning (see [Chapter 6](#))

---

## 3.3 Function Prefixes

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

- **ZLL functions:** Used to interact with the ZLL profile and prefixed with **xZLL\_**
- **ZCL functions:** Used to interact with the ZCL and prefixed with **xZCL\_**
- **Cluster functions:** Used to interact with clusters and prefixed as follows:
  - For clusters defined in the ZLL specification, they are prefixed with **xZLL\_**
  - For clusters defined in the ZCL specification, they are prefixed with **xCLD\_**

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

Only functions that are ZLL-specific are detailed in this manual. Functions which relate to clusters of the ZCL are detailed in the *ZCL User Guide (JN-UG-3103)*.

---

## 3.4 Development Phases

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



**Note:** Before starting your ZLL application development, you should familiarise yourself with the general aspects of ZigBee PRO application development, described in the *ZigBee PRO Stack User Guide (JN-UG-3101)*.

1. **Network Configuration:** Configure the ZigBee network parameters for the nodes using the ZPS Configuration Editor - refer to the *ZigBee PRO Stack User Guide (JN-UG-3101)*.
2. **OS Configuration:** Configure the JenOS resources to be used by your application using the JenOS Configuration Editor - refer to the *JenOS User Guide (JN-UG-3075)*.
3. **Application Code Development:** Develop the application code for your nodes using the ZigBee PRO APIs, JenOS APIs, ZLL API and ZCL - refer to the *ZigBee PRO Stack User Guide (JN-UG-3101)*, *JenOS User Guide (JN-UG-3075)* and *ZCL User Guide (JN-UG-3103)*, as well as this manual.
4. **Application Build:** Build the application binaries for your nodes using the JN51xx compiler and linker built into BeyondStudio for NXP - refer to [Section 3.5](#) and to the *BeyondStudio for NXP Installation and User Guide (JN-UG-3098)*.
5. **Node Programming:** Load the application binaries into Flash memory on your nodes using the JN516x Flash programmer built into BeyondStudio for NXP - refer to the *BeyondStudio for NXP Installation and User Guide (JN-UG-3098)*.

---

## 3.5 Building an Application

This section outlines how to build a ZLL application developed for the JN516x device. First of all, the configuration of compile-time options and ZigBee network parameters is described, and then directions are given for building and loading the application.

---

### 3.5.1 Compile-Time Options

Before the application can be built, the ZLL compile-time options must be configured in the header file **zcl\_options.h** for the application. This header file is supplied in the Application Note *ZigBee Light Link Solution (JN-AN-1171)*, which can be used as a template.

#### Number of Endpoints

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

```
#define ZLL_NUMBER_OF_ENDPOINTS 3
```

Normally, the endpoints starting at endpoint 1 will be used for ZLL, so in the above case endpoints 1 to 3 will be used for ZLL. It is possible, however, to use the lower numbered endpoints for non-ZLL purposes, e.g. to run other protocols on endpoints 1 and 2, and ZLL on endpoint 3. In this case, with `ZLL_NUMBER_OF_ENDPOINTS` set to 3, some storage will be statically allocated by ZLL 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 `ZLL_NUMBER_OF_ENDPOINTS`.

#### 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
#define CLD_LEVEL_CONTROL
#define CLD_ZLL_COMMISSION
```

#### Server and Client Options

Many clusters used in ZLL 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 definition 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
```



**Note:** Cluster-specific compile-time options are detailed in the chapters for the individual clusters in [Part II: ZLL Clusters](#). For clusters from the ZCL, refer to the *ZCL User Guide (JN-UG-3103)*.

---

## 3.5.2 ZigBee Network Parameters

ZLL applications may require specific settings of certain ZigBee network parameters. These parameters are set using the ZPS Configuration Editor. The full set of ZigBee network parameters are detailed in the *ZigBee PRO Stack User Guide (JN-UG-3101)*.

---

## 3.5.3 Building and Loading the Application Binary

A ZLL application for the JN516x device is built within BeyondStudio for NXP. For instructions on building an application, refer to the *BeyondStudio for NXP Installation and User Guide (JN-UG-3098)*. This guide also indicates how to load the built application binary file into a JN516x device using the integrated JN516x Flash programmer.

**Chapter 3**  
**ZLL Application Development**

---

## 4. ZLL Application Coding

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



**Note:** ZCL API functions referenced in this chapter are fully described in the *ZCL User Guide (JN-UG-3103)*.

---

### 4.1 ZLL Programming Concepts

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

---

#### 4.1.1 Shared Device Structures

In each ZLL device, attribute values are exchanged between the application and the ZLL library by means of a shared structure. This structure is protected by a mutex (described in the *ZCL User Guide (JN-UG-3103)*). The structure for a particular ZLL device contains structures for the clusters supported by that device (see [Chapter 2](#)). The available device structures are detailed in [Section 8.1](#).



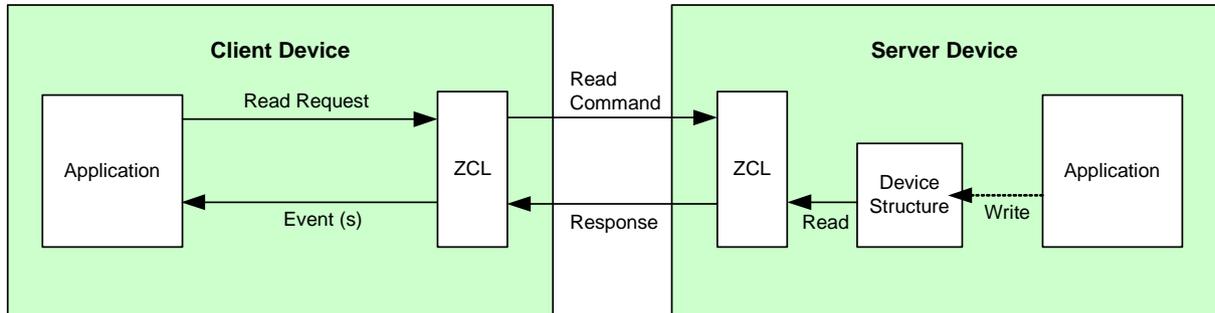
**Note:** In order to use a cluster which is supported by a device, the relevant option for the cluster must be specified at build-time - see [Section 3.5.1](#).

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

- The local application writes attribute values to the structure, allowing the ZigBee Cluster Library (ZCL) to respond to commands relating to these attributes.
- The ZCL parses incoming commands that write attribute values to the structure. The written values can then be read by the local application.

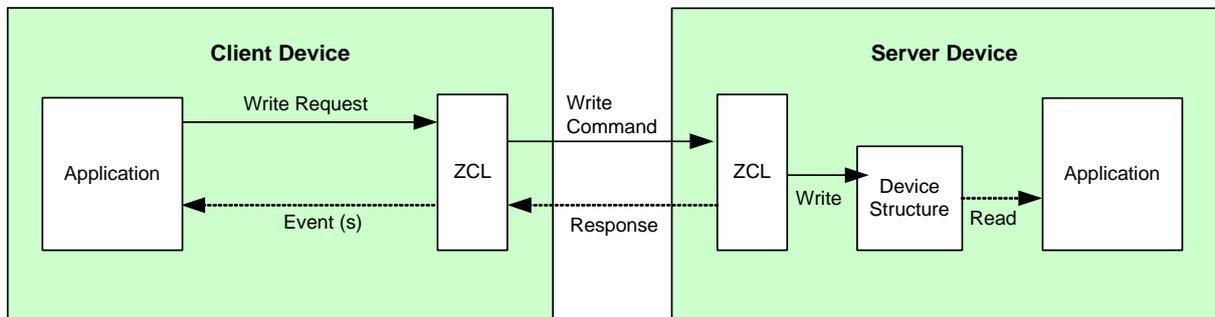
Remote read and write operations involving a shared device structure are illustrated in [Figure 3](#) below. For more detailed descriptions of these operations, refer to [Section 4.5](#) and [Section 4.6](#).

### Reading Remote Attributes



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

### Writing Remote Attributes



1. ZCL sends 'write attributes' request to remote server.
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.
5. ZCL can receive optional response and generate events for the application (that indicate any unsuccessful writes).

**Figure 3: Operations using Shared Device Structure**



**Note:** If there are no remote writes to the attributes of a cluster server, these attributes are maintained only by the local application(s).

---

## 4.1.2 Addressing

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

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

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

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

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

A destination address structure, `tsZCL_Address`, is defined in the ZCL and is detailed in the *ZCL User Guide (JN-UG-3103)*. Enumerations are provided for the addressing mode and broadcast mode in this structure, and are also detailed in the above manual.

---

## 4.1.3 OS Resources

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

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

A JenOS mutex protects the shared structure that holds the cluster attribute values for a device (see [Section 4.1.1](#) above). The ZCL invokes an application callback function to lock and unlock this mutex. The mutex should be used in conjunction with the counting mutex code provided in the appendix of the *ZCL User Guide (JN-UG-3103)*.

The software for this mutex operation is contained in the ZLL demonstration application.

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

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

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

---

## 4.2 Initialisation

A ZLL application is initialised like a normal ZigBee PRO application, as described in the section “Forming a Network” of the *ZigBee PRO Stack User Guide (JN-UG-3101)*, except there is no need to explicitly start the ZigBee PRO stack using the function **ZPS\_eAplZdoStartStack()**. In addition, some ZLL initialisation must be performed in the application code.

The ZLL initialisation functions mentioned below must be called after calling **ZPS\_eAplAfnit()**:

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



**Note:** The set of endpoint registration functions for the different ZLL device types are detailed in [Chapter 7](#).

---

## 4.3 Callback Functions

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

- **Endpoint Callback Function:** A callback function must be provided for each endpoint used, where this callback function will be invoked when an event occurs (such as an incoming message) relating to the endpoint. The callback function is registered with the ZLL library when the endpoint is registered using the registration function for the ZLL device type that the endpoint supports - for example, using **eZLL\_RegisterOnOffLightEndPoint()** for an On/Off Light device (see [Chapter 7](#)).
- **General Callback Function:** Events that do not have an associated endpoint are delivered via a callback function that is registered with the ZLL library through the function **eZLL\_Initialise()**. For example, stack leave and join events can be received by the application through this callback function.

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

```
typedef void (* tfpZCL_ZCLCallbackFunction)  
              (tsZCL_CallbackEvent *pCallbackEvent);
```

The callback events are detailed in the *ZCL User Guide (JN-UG-3103)* and event handling is further described in [Section 4.7](#).

---

## 4.4 Network Formation/Joining

The formation of a ZLL network is handled by the Touchlink feature (see [Section 1.4](#)). A node is added to the network using a special node called an 'initiator', which is usually a remote control unit. Touchlink uses the ZLL Commissioning cluster, which is fully detailed in [Chapter 6](#) (Touchlink installation is described in [Section 6.4](#)).

As part of the Touchlink installation of a node, the initiator obtains the following information from the joining node:

- endpoint number of ZLL application
- device type
- network address
- IEEE/MAC address

The initiator stores this information in an endpoint table for the application.

---

## 4.5 Reading Attributes

Attributes can be read using a general ZCL function, or using a ZLL or ZCL function which is specific to the target cluster. The cluster-specific functions for reading attributes are covered in the chapters of this manual that describe the supported clusters or in the *ZCL User Guide (JN-UG-3103)*. Note that read access to cluster attributes must be explicitly enabled at compile-time as described in [Section 3.5.1](#).

The remainder of this section describes the use of the ZCL function **eZCL\_SendReadAttributesRequest()** to send a 'read attributes' request, although the sequence is similar when using the cluster-specific 'read attributes' functions. The resulting activities on the source and destination nodes are outlined below and illustrated in [Figure 4](#). The events generated from a 'read attributes' request are further described in [Section 4.7](#).

### 1. On Source Node (Client)

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

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

### 2. On Destination Node (Server)

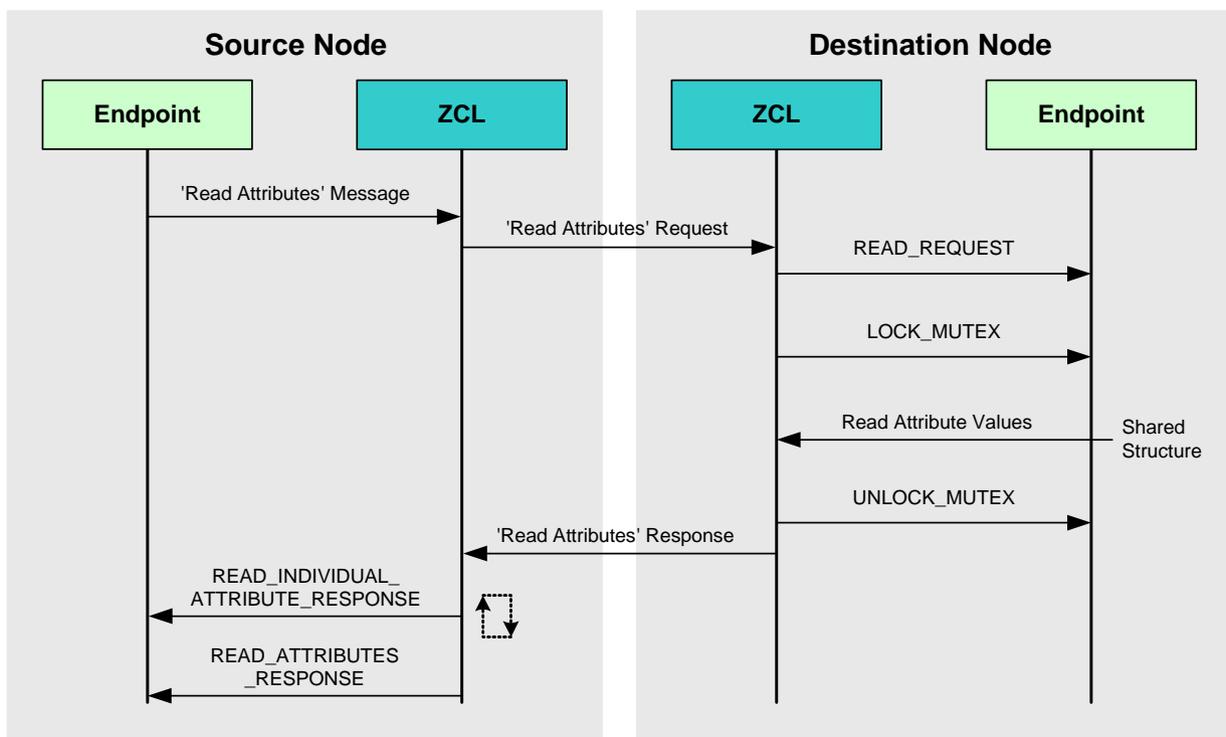
On receiving the 'read attributes' request, the ZCL software on the destination node performs the following steps:

1. Generates an `E_ZCL_CBET_READ_REQUEST` event for the destination endpoint callback function which, if required, can update the shared device structure that contains the attributes to be read, before the read takes place.
2. Generates an `E_ZCL_CBET_LOCK_MUTEX` event for the endpoint callback function, which should lock the mutex that protects the shared device structure - for information on mutexes, refer to the *ZCL User Guide (JN-UG-3103)*
3. Reads the relevant attribute values from the shared device structure and creates a 'read attributes' response message containing the read values.
4. Generates an `E_ZCL_CBET_UNLOCK_MUTEX` event for the endpoint callback function, which should now unlock the mutex that protects the shared device structure (other application tasks can now access the structure).
5. Sends the 'read attributes' response to the source node of the request.

### 3. On Source Node (Client)

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

1. For each attribute listed in the 'read attributes' response, it generates an E\_ZCL\_CBET\_READ\_INDIVIDUAL\_ATTRIBUTE\_RESPONSE message for the source endpoint callback function, which may or may not take action on this message.
2. On completion of the parsing of the 'read attributes' response, it generates a single E\_ZCL\_CBET\_READ\_ATTRIBUTES\_RESPONSE message for the source endpoint callback function, which may or may not take action on this message.



**Figure 4: 'Read Attributes' Request and Response**



**Note:** The 'read attributes' requests and responses arrive at their destinations as data messages. Such a message triggers a stack event of the type ZPS\_EVENT\_APS\_DATA\_INDICATION, which is handled as described in [Section 4.7](#).

---

## 4.6 Writing Attributes

The ability to write attribute values to a remote cluster is required by ZLL controller devices. Normally, a 'write attributes' request is sent from a client cluster to a server cluster, where the relevant attributes in the shared device structure are updated. Note that write access to cluster attributes must be explicitly enabled at compile-time as described in [Section 3.5.1](#).

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

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

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

The activities surrounding a 'write attributes' request on the source and destination nodes are outlined below and illustrated in [Figure 5](#). The events generated from a 'write attributes' request are further described in [Section 4.7](#).

### 1. On Source Node (Client)

In order to send a 'write attributes' request, the application on the source node calls one of the above ZCL 'write attributes' functions to submit a request to update the relevant attributes on a cluster on a remote node. The information required by this function includes the following:

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

## 2. On Destination Node (Server)

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

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

In the case of an out-of-range value or a blocked write, there is no further processing for that particular attribute following the 'write attributes' request.

2. Generates an `E_ZCL_CBET_LOCK_MUTEX` event for the endpoint callback function, which should lock the mutex that protects the relevant shared device structure - for more on mutexes, refer to the *ZCL User Guide (JN-UG-3103)*.
3. Writes the relevant attribute values to the shared device structure - an `E_ZCL_CBET_WRITE_INDIVIDUAL_ATTRIBUTE` event is generated for each individual attempt to write an attribute value, which the endpoint callback function can use to keep track of the successful and unsuccessful writes.

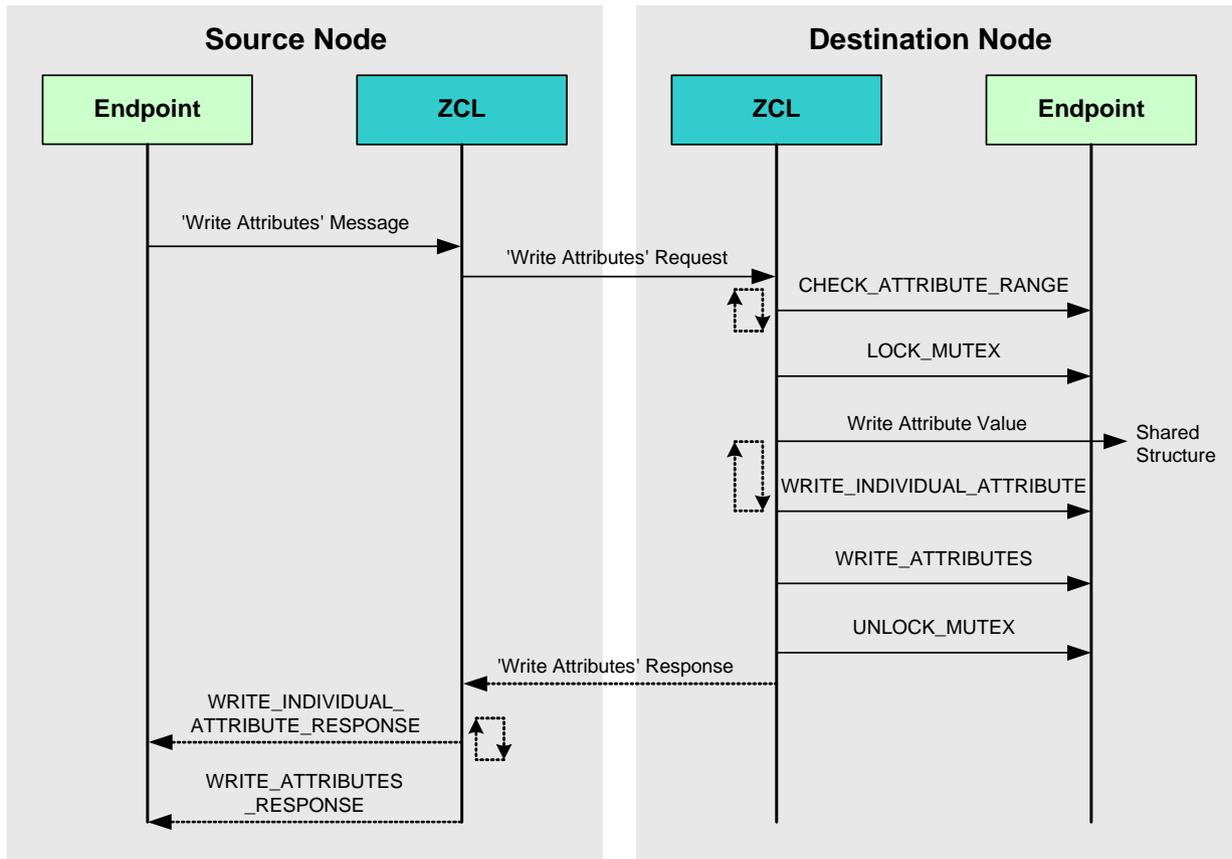
Note that if an 'undivided write attributes' request was received, an individual failed write will render the whole update process unsuccessful.

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

## 3. On Source Node (Client)

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

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



**Figure 5: 'Write Attributes' Request and Response**

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

## 4.7 Handling Stack and Timer Events

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

The event handling framework for ZigBee Light Link is provided by the ZCL. The event must be wrapped in a `tsZCL_CallbackEvent` structure by the application, which then passes this event structure into the ZCL using the function **vZCL\_EventHandler()**. The ZCL processes the event and, if necessary, invokes the relevant endpoint callback function. This event structure and event handler function are detailed in the *ZCL User Guide (JN-UG-3103)*, which also provides more details of event processing.

The events that are not cluster-specific are divided into four categories, as shown in [Table 17](#) below - these events are described in the *ZCL User Guide (JN-UG-3103)*. Cluster-specific events are covered in the chapter for the relevant cluster.

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

**Table 17: Events (Not Cluster-Specific)**



**Note:** ZCL error events and default responses may be generated when problems occur in receiving commands. The possible ZCL status codes contained in the events and responses are detailed in the *ZCL User Guide (JN-UG-3103)*.

## 4.8 Servicing Timing Requirements

Some clusters used by a ZLL application may have timing requirements which demand periodic updates. The function **eZLL\_Update100mS()** is provided to service these requirements and should be called repeatedly every 100 ms. Invocation of this function can be prompted using a 100-ms software timer.

The function **eZLL\_Update100mS()** calls the external function **vldEffectTick()**, which must be defined in the application. This user-defined function can be used to implement an identify effect on the node, if required. Otherwise, it should be defined but left empty.

# Part II: ZLL Clusters



---

## 5. ZCL Clusters

The ZLL application profile uses certain clusters that are provided in the ZigBee Cluster Library (ZCL):

- Basic - see [Section 5.1](#)
- Identify - see [Section 5.2](#)
- Groups - see [Section 5.3](#)
- Scenes - see [Section 5.4](#)
- On/Off - see [Section 5.5](#)
- Level Control - see [Section 5.6](#)
- Colour Control - see [Section 5.7](#)

These clusters are briefly introduced below and are fully detailed in the *ZCL User Guide (JN-UG-3103)*.



**Note:** The above clusters contain special enhancements for ZigBee Light Link but can also be used with other ZigBee application profiles, such as Home Automation.

---

## 5.1 Basic Cluster

The Basic cluster is mandatory for all ZLL devices as a server-side cluster. It has a Cluster ID of 0x0000.

---

### 5.1.1 Mandatory Attributes for ZLL

The following Basic cluster server-side attributes are mandatory for ZigBee Light Link:

Attribute	Structure Field(s)
ZCLVersion	u8ZCLVersion
ApplicationVersion	u8ApplicationVersion
StackVersion	u8StackVersion
HWVersion	u8HardwareVersion
ManufacturerName	sManufacturerName au8ManufacturerName[32]
ModelIdentifier	sModelIdentifier au8ModelIdentifier[32]
DateCode	sDateCode au8DateCode[16]
PowerSource	ePowerSource
SWBuildID *	sSWBuildID au8SWBuildID[16]

**Table 18: Mandatory Server-side Attributes**

\* This is an additional attribute for ZigBee Light Link

---

### 5.1.2 Compile-Time Options

You must include the header file **Basic.h** in your application.

The Basic cluster is enabled in the **zcl\_options.h** file by means of the definition:

```
#define CLD_BASIC
```

In addition, you must enable the cluster as a server using:

```
#define BASIC_SERVER
```

The ZLL-specific attribute SWBuildID can be enabled using:

```
#define CLD_BAS_ATTR_SW_BUILD_ID
```

Other compile-time options are also available for the Basic cluster and are described in the *ZCL User Guide (JN-UG-3103)*.

---

## 5.2 Identify Cluster

The Identify cluster allows a device to identify itself (for example, by flashing a LED on the node).

It has a Cluster ID of 0x0003.

---

### 5.2.1 Mandatory Attribute for ZLL

The following Identify cluster server-side attribute is mandatory for ZigBee Light Link:

Attribute	Structure Field
IdentifyTime	u16IdentifyTime

**Table 19: Mandatory Server-side Attribute**

---

### 5.2.2 Enhanced Functionality for ZLL

The Identify cluster contains extra functionality for ZLL. This is the 'Trigger Effect' command - a function is provided to issues this command. This feature and the associated function are described in the *ZCL User Guide (JN-UG-3103)*.

---

### 5.2.3 Compile-Time Options

To use the Identify cluster, you must include the header file **Identify.h** in your application.

The Identify cluster is enabled in the **zcl\_options.h** file by means of the definition:

```
#define CLD_IDENTIFY
```

In addition, you must enable the cluster as a server or client, using one of:

```
#define IDENTIFY_SERVER  
#define IDENTIFY_CLIENT
```

To enable the enhanced cluster functionality for ZLL (see [Section 5.2.2](#)), you must include:

```
#define CLD_IDENTIFY_SUPPORT_ZLL_ENHANCED_COMMANDS
```

---

## 5.3 Groups Cluster

The Groups cluster allows the management of the Group table concerned with group addressing.

It has a Cluster ID of 0x0004.

---

### 5.3.1 Mandatory Attribute for ZLL

The following Groups cluster server-side attribute is mandatory for ZigBee Light Link:

Attribute	Structure Field
NameSupport	u8NameSupport

**Table 20: Mandatory Server-side Attribute**

Name support must be disabled for ZLL by setting the NameSupport attribute to zero. This setting can be pre-configured at compile-time - see [Section 5.3.2](#) below.

---

### 5.3.2 Compile-Time Options

To use the Groups cluster, you must include the header file **Groups.h** in your application.

The Groups cluster is enabled in the **zcl\_options.h** file by means of the definition:

```
#define CLD_GROUPS
```

In addition, you must enable the cluster as a server or client, using one of:

```
#define GROUPS_SERVER  
#define GROUPS_CLIENT
```

Name support must be disabled for ZLL, which can be done using:

```
#define CLD_GROUPS_DISABLE_NAME_SUPPORT
```

---

## 5.4 Scenes Cluster

The Groups cluster allows the management of ‘scenes’, where a scene corresponds to particular level settings for a set of lights (usually in a group).

It has a Cluster ID of 0x0005.

---

### 5.4.1 Mandatory Attributes for ZLL

The following Scenes cluster server-side attributes are mandatory for ZigBee Light Link:

Attribute	Structure Field
SceneCount	u8SceneCount
CurrentScene	u8CurrentScene
CurrentGroup	u16CurrentGroup
SceneValid	bSceneValid
NameSupport	u8NameSupport
TransitionTime100ms *	u8TransitionTime100ms

**Table 21: Mandatory Server-side Attributes**

\* This is an additional attribute for ZigBee Light Link

Name support must be disabled for ZLL by setting the NameSupport attribute to zero. This setting can be pre-configured at compile-time - see [Section 5.4.3](#) below.

---

### 5.4.2 Enhanced Functionality for ZLL

The Scenes cluster contains extra functionality for ZLL. This is the ‘Copy Scene’ command - a function is provided to issue this command. This feature and the associated function are described in the *ZCL User Guide (JN-UG-3103)*.

---

### 5.4.3 Compile-Time Options

To use the Scenes cluster, you must include the header file **Scenes.h** in your application.

The Scenes cluster is enabled in the **zcl\_options.h** file by means of the definition:

```
#define CLD_SCENES
```

In addition, you must enable the cluster as a server or client, using one of:

```
#define SCENES_SERVER
```

```
#define SCENES_CLIENT
```

## **Chapter 5**

### **ZCL Clusters**

To enable the enhanced cluster functionality for ZLL (see [Section 5.4.2](#)), you must include:

```
#define CLD_SCENES_SUPPORT_ZLL_ENHANCED_COMMANDS
```

Name support must be disabled for ZLL, which can be done using:

```
#define CLD_SCENES_DISABLE_NAME_SUPPORT
```

---

## 5.5 On/Off Cluster

The On/Off cluster allows allows a device to be put into the 'on' and 'off' states, or toggled between the two states.

It has a Cluster ID of 0x0006.

---

### 5.5.1 Mandatory Attributes for ZLL

The following On/Off cluster server-side attributes are mandatory for ZigBee Light Link:

Attribute	Structure Field
OnOff	bOnOff
GlobalSceneControl *	bGlobalSceneControl
OnTime *	u16OnTime
OffWaitTime *	u16OffWaitTime

**Table 22: Mandatory Server-side Attributes**

\* These are additional attributes for ZigBee Light Link

---

### 5.5.2 Enhanced Functionality for ZLL

The On/Off cluster contains extra functionality for ZLL. This includes the 'Off With Effect' and 'On With Timed Off' commands - functions are provided to issue these commands. In addition, a facility is provided to save the current lights settings when the lights are switched off (and recall the settings when the lights are switched on). These features and the associated functions are described in the *ZCL User Guide (JN-UG-3103)*.

---

### 5.5.3 Compile-Time Options

To use the On/Off cluster, you must include the header file **OnOff.h** in your application.

The On/Off cluster is enabled in the **zcl\_options.h** file by means of the definition:

```
#define CLD_ONOFF
```

In addition, you must enable the cluster as a server or client, using one of:

```
#define ONOFF_SERVER  
#define ONOFF_CLIENT
```

## Chapter 5

### ZCL Clusters

To enable the enhanced cluster functionality for ZLL (see [Section 5.5.2](#)), you must include:

```
#define CLD_ONOFF_SUPPORT_ZLL_ENHANCED_COMMANDS
```

The ZLL-specific attribute GlobalSceneControl can be enabled using:

```
#define CLD_ONOFF_ATTR_GLOBAL_SCENE_CONTROL
```

The ZLL-specific attribute OnTime can be enabled using:

```
#define CLD_ONOFF_ATTR_ON_TIME
```

The ZLL-specific attribute OffWaitTime can be enabled using:

```
#define CLD_ONOFF_ATTR_OFF_WAIT_TIME
```

---

## 5.6 Level Control Cluster

The Level Control cluster allows control of the level of a physical quantity on a device, where this physical quantity is device-dependent - in the case of ZLL, it is normally light level.

It has a Cluster ID of 0x0008.

---

### 5.6.1 Mandatory Attributes for ZLL

The following Level Control cluster server-side attributes are mandatory for ZigBee Light Link:

Attribute	Structure Field
CurrentLevel	u8CurrentLevel
RemainingTime	u16RemainingTime

**Table 23: Mandatory Server-side Attributes**

---

### 5.6.2 Compile-Time Options

To use the Level Control cluster, you must include the header file **LevelControl.h** in your application.

The Level Control cluster is enabled in the **zcl\_options.h** file by means of the definition:

```
#define CLD_LEVEL_CONTROL
```

In addition, you must enable the cluster as a server or client, using one of:

```
#define LEVEL_CONTROL_SERVER  
#define LEVEL_CONTROL_CLIENT
```

## 5.7 Colour Control Cluster

The Colour Control cluster allows the colour of a light to be controlled (note that it does not govern the overall luminance of the light, as this is controlled using the Level Control cluster).

It has a Cluster ID of 0x0300.

### 5.7.1 Mandatory Attributes for ZLL

The following Colour Control cluster server-side attributes are mandatory for ZigBee Light Link:

Attribute	Structure Field
CurrentHue	u8CurrentHue
CurrentSaturation	u8CurrentSaturation
RemainingTime	u16RemainingTime
CurrentX	u16CurrentX
CurrentY	u16CurrentY
ColorTemperature (mired)	u16ColourTemperatureMired
ColorMode	u8ColourMode
NumberOfPrimaries	u8NumberOfPrimaries
Primary1X	u16Primary1X
Primary1Y	u16Primary1Y
Primary1Intensity	u8Primary1Intensity
Primary2X	u16Primary2X
Primary2Y	u16Primary2Y
Primary2Intensity	u8Primary2Intensity
Primary3X	u16Primary3X
Primary3Y	u16Primary3Y
Primary3Intensity	u8Primary3Intensity
Primary4X	u16Primary4X
Primary4Y	u16Primary4Y
Primary4Intensity	u8Primary4Intensity
Primary5X	u16Primary5X
Primary5Y	u16Primary5Y

**Table 24: Mandatory Server-side Attributes**

Attribute	Structure Field
Primary5Intensity	u8Primary5Intensity
Primary6X	u16Primary6X
Primary6Y	u16Primary6Y
Primary6Intensity	u8Primary6Intensity
EnhancedCurrentHue *	u16EnhancedCurrentHue
EnhancedColorMode *	u8EnhancedColourMode
ColorLoopActive *	u8ColourLoopActive
ColorLoopDirection *	u8ColourLoopDirection
ColorLoopTime *	u16ColourLoopTime
ColorLoopStartEnhancedHue *	u16ColourLoopStartEnhancedHue
ColorLoopStoredEnhancedHue *	u16ColourLoopStoredEnhancedHue
ColorCapabilities *	u16ColourCapabilities
ColorTempPhysicalMin *	u16ColourTemperatureMiredPhyMin
ColorTempPhysicalMax *	u16ColourTemperatureMiredPhyMax

**Table 24: Mandatory Server-side Attributes**

\* These are additional attributes for ZigBee Light Link

## 5.7.2 Enhanced Functionality for ZLL

The Colour Control cluster contains extra functionality for ZLL. This includes all the enhanced hue commands ('Enhanced Move to Hue', 'Enhanced Move Hue' and 'Enhanced Step Hue'), the 'Move Colour Temperature' and 'Step Colour Temperature' commands, and the 'Colour Loop Set' command - functions are provided to issue these commands. These features and the associated functions are described in the *ZCL User Guide (JN-UG-3103)*. Compile-time options for the enhanced functionality are detailed in [Section 5.7.3](#).

### 5.7.3 Compile-Time Options

To use the Colour Control cluster, you must include the header file **ColourControl.h** in your application.

The Colour Control cluster is enabled in the **zcl\_options.h** file by means of the definition:

```
#define CLD_COLOUR_CONTROL
```

In addition, you must enable the cluster as a server or client, using one of:

```
#define COLOUR_CONTROL_SERVER
#define COLOUR_CONTROL_CLIENT
```

To enable the enhanced cluster functionality for ZLL (see [Section 5.7.2](#)), you must include:

```
#define CLD_COLOURCONTROL_ATTR_ENHANCED_COLOUR_MODE
#define CLD_COLOURCONTROL_ATTR_COLOUR_CAPABILITIES
```

If required, the ZLL enhanced attributes must be enabled through a 'Colour Capabilities' definition. Attributes are enabled as a group according to the required capability/functionality. The capabilities are detailed in the table below, with their corresponding attributes and macros.

Capability/Functionality	Attributes	Macro
Hue/Saturation	u8CurrentHue* u8CurrentSaturation*	COLOUR_CAPABILITY_HUE_SATURATION_SUPPORTED
Enhanced Hue (also need Hue/Saturation)	u16EnhancedCurrentHue	COLOUR_CAPABILITY_ENHANCE_HUE_SUPPORTED
Colour Loop (also need Enhanced Hue)	u8ColourLoopActive u8ColourLoopDirection u16ColourLoopTime u16ColourLoopStartEnhancedHue u16ColourLoopStoredEnhancedHue	COLOUR_CAPABILITY_COLOUR_LOOP_SUPPORTED
CIE XY Values (this is mandatory)	u16CurrentX* u16CurrentY*	COLOUR_CAPABILITY_XY_SUPPORTED
Colour Temperature	u16ColourTemperatureMired u16ColourTemperatureMiredPhyMin u16ColourTemperatureMiredPhyMax	COLOUR_CAPABILITY_COLOUR_TEMPERATURE_SUPPORTED

**Table 25: 'Colour Capabilities' Macros**

\* These attributes are not ZLL enhanced attributes

The above macros will automatically invoke the macros for the individual attributes in the capability group, e.g. `CLD_COLOURCONTROL_ATTR_COLOUR_LOOP_TIME` for the attribute `u16ColourLoopTime`.

The enabled Colour Capabilities are reflected in the ZLL enhanced attribute (bitmap) `ul6ColourCapabilities`.

Example Colour Capabilities definitions are provided below for different ZLL devices.

**ZLL Extended Colour Light:**

```
#define CLD_COLOURCONTROL_COLOUR_CAPABILITIES
    ( COLOUR_CAPABILITY_HUE_SATURATION_SUPPORTED | \
      COLOUR_CAPABILITY_ENHANCE_HUE_SUPPORTED   | \
      COLOUR_CAPABILITY_COLOUR_LOOP_SUPPORTED   | \
      COLOUR_CAPABILITY_XY_SUPPORTED             | \
      COLOUR_CAPABILITY_COLOUR_TEMPERATURE_SUPPORTED)
```

**ZLL Color Light:**

```
#define CLD_COLOURCONTROL_COLOUR_CAPABILITIES
    ( COLOUR_CAPABILITY_HUE_SATURATION_SUPPORTED | \
      COLOUR_CAPABILITY_ENHANCE_HUE_SUPPORTED   | \
      COLOUR_CAPABILITY_COLOUR_LOOP_SUPPORTED   | \
      COLOUR_CAPABILITY_XY_SUPPORTED)
```

**ZLL Color Temperature Light:**

```
#define CLD_COLOURCONTROL_COLOUR_CAPABILITIES
    ( COLOUR_CAPABILITY_COLOUR_TEMPERATURE_SUPPORTED)
```

**Chapter 5**  
**ZCL Clusters**

## 6. ZLL Commissioning Cluster

This chapter describes the ZLL Commissioning cluster which is defined in the ZigBee Light Link profile, and is used when forming a ZLL network or adding a new node to an existing ZLL network. This cluster facilitates the Touchlink feature of ZigBee Light Link (see [Section 1.4](#)).

The ZLL Commissioning cluster has a Cluster ID of 0x1000.

### 6.1 Overview

The ZLL Commissioning cluster is associated with a ZLL node as a whole, rather than with individual ZLL devices on the node. It must be used on nodes that incorporate one or more of the ZLL devices indicated in [Table 26](#) below, which shows the supported devices when the ZLL Commissioning cluster acts as a client, server and combined client/server.

Client	Client/Server	Server
Colour Controller	Colour Controller	Colour Controller
Colour Scene Controller	Colour Scene Controller	Colour Scene Controller
Non-Colour Controller	Non-Colour Controller	Non-Colour Controller
Non-Colour Scene Controller	Non-Colour Scene Controller	Non-Colour Scene Controller
Control Bridge	Control Bridge	Control Bridge
On/Off Sensor	On/Off Sensor	On/Off Sensor
	On/Off Light	On/Off Light
	On/Off Plug-in Unit	On/Off Plug-in Unit
	Dimmable Light	Dimmable Light
	Dimmable Plug-in Unit	Dimmable Plug-in Unit
	Colour Light	Colour Light
	Extended Colour Light	Extended Colour Light
	Colour Temperature Light	Colour Temperature Light

**Table 26: ZLL Commissioning Cluster in Devices**

This cluster supports two sets of functionality, corresponding to two distinct commands sets:

- Touchlink
- Commissioning Utility

The ZLL API provides functions for implementing both sets of commands. These functions are referenced in [Section 6.4](#) and [Section 6.5](#), and detailed in [Section 6.7](#).

The Commissioning Utility functionality is not required on ZLL Lighting devices.

For the compile-time options for enabling the ZLL Commissioning cluster for Touchlink and the Commissioning Utility, refer to [Section 6.10](#).

---

## 6.2 ZLL Commissioning Cluster Structure and Attributes

This cluster has no attributes, as a server or a client. Therefore, the cluster structure `tsCLD_ZllCommission` is referred to using a null pointer.

---

## 6.3 Commissioning Operations

ZLL commissioning involves forming a ZLL network or adding a new node to an existing ZLL network. A node from which commissioning can be initiated is referred to as an 'initiator' - this may be a remote control unit, but could also be a lamp.

- An 'initiator' node must support the ZLL Commissioning cluster as a client.
- A node to be added to the network must support the ZLL Commissioning cluster as a server (or as both a server and client).

Note that commissioning a new network involves adding at least one node to the new network (as well as the initiator).

Commissioning may involve two stages, depending on the type of node added to the network by the initiator:

1. The node is added to the network using the Touchlink commands of the ZLL Commissioning cluster. In practice for the user, this typically involves bringing the initiator node physically close to the target node and pressing a button.
2. If the initiator node and the new node will both be used to control lights in the network, the new node must learn certain information (such as controlled endpoints and configured groups) from the initiator. This exchange of information uses the Commissioning Utility commands of the ZLL Commissioning cluster.



**Note:** The ZLL Commissioning cluster instance for Touchlink must reside on its own endpoint on a node. Therefore, a Touchlink commissioning application must be provided which is distinct from the main ZLL application. However, the cluster instance for the Commissioning Utility can reside on the same endpoint as the main application (and be used in this application).

Commissioning using the supplied functions for Touchlink and the Commissioning Utility is described in [Section 6.4](#) and [Section 6.5](#).

## 6.4 Using Touchlink

Touchlink is used for the basic commissioning of a new network or adding a new node to an existing network. A dedicated Touchlink application (which is distinct from the main ZLL application on the node) must reside on its own endpoint. This requires:

- a ZLL Commissioning cluster instance as a client to be created on the endpoint on the initiator node
- a ZLL Commissioning cluster instance as a server to be created on the endpoint on the target node

The initiator node will also require a ZLL Commissioning cluster instance as a server (on the same endpoint), since the node also needs the capability to join an existing ZLL network.

An endpoint is registered for Touchlink (on both nodes) using the function **eZLL\_RegisterCommissionEndPoint()**. This function also creates a ZLL Commissioning cluster instance of the type (server, client or both) determined by the compile-time options in the header file **zcl\_options.h** (see [Section 6.10](#)).

The initiator must then send a sequence of request commands to the target node. The Touchlink request command set is summarised in [Table 27](#). Touchlink functions for issuing these commands are provided in the ZLL API and detailed in [Section 6.7.1](#).

Command	Identifier	Description
Scan Request *	0x00	Requests other devices (potential nodes) in the local neighbourhood to respond. A scan request is first performed on channel 11, up to five times until a response is received. If no response is received, a scan request is then performed once on each of channels 15, 20 and 25, and then the remaining channels (12, 13, 14, 16, etc) until a response is detected.
Device Information Request *	0x02	Requests information about the devices on a remote node
Identify Request	0x06	Requests a remote node to physically identify itself (e.g. visually by flashing a LED)
Reset To Factory New Request	0x07	Requests a factory reset of a remote node
Network Start Request *	0x10	Requests a new network to be created comprising the initiator and a detected Router
Network Join Router Request *	0x12	Requests a Router to join the network
Network Join End Device Request *	0x14	Requests an End Device to join the network
Network Update Request *	0x16	Requests an update of the network settings on a remote node (if the supplied Network Update Identifier is more recent than the one on the node)

**Table 27: Touchlink Request Commands**

\* These commands have corresponding responses.

All Touchlink commands are sent as inter-PAN messages.

Use of the above commands and associated functions is described in the sub-sections below.

---

## 6.4.1 Creating a ZLL Network

A ZLL network is formed from an initiator node and a Router node (usually the initiator is an End Device and will have no routing capability in the network). The Touchlink network creation process is described below and is illustrated in [Figure 6](#) (also refer to the command list in [Table 27 on page 69](#)).

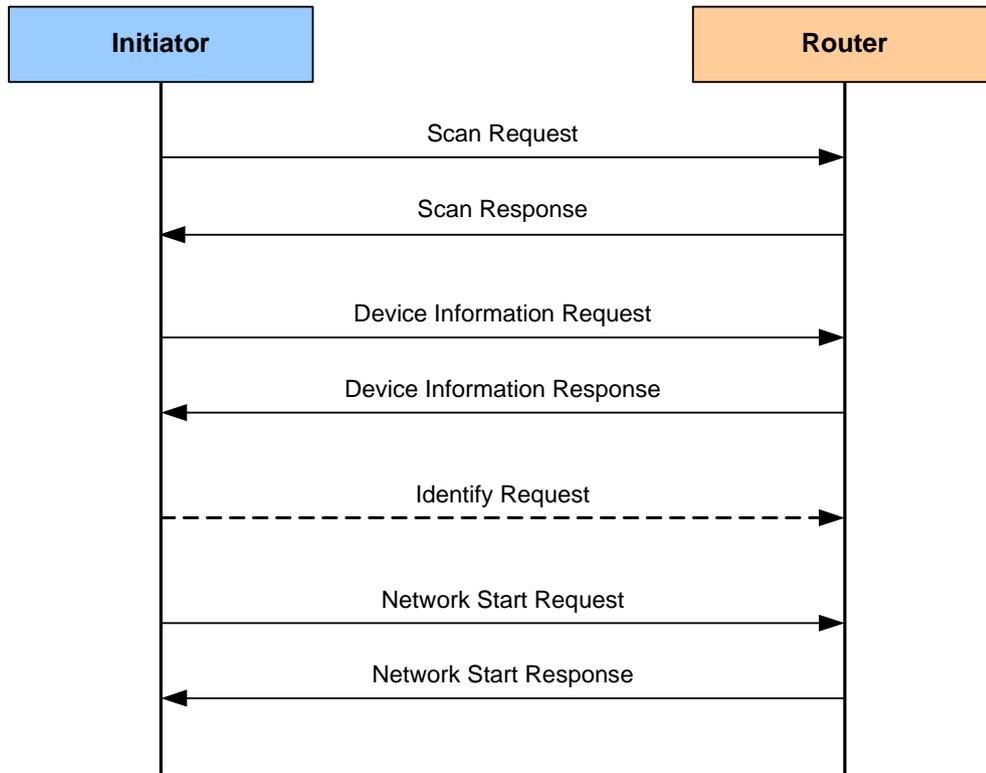


**Note:** Received Touchlink requests and responses are handled as ZigBee PRO events. The event handling is not detailed below but is outlined in [Section 6.6](#).

- 1. Scan Request:** The initiator sends a Scan Request to nodes in its vicinity. The required function is:  
`eCLD_ZIICommissionCommandScanReqCommandSend()`
- 2. Scan Response:** A receiving node replies to the Scan Request by sending a Scan Response, which includes the device type of the responding node (e.g. Router). The required function is:  
`eCLD_ZIICommissionCommandScanRspCommandSend()`
- 3. Device Information Request:** The initiator sends a Device Information Request to the detected Routers that are of interest. The required function is:  
`eCLD_ZIICommissionCommandDeviceInfoReqCommandSend()`
- 4. Device Information Response:** A receiving Router replies to the Device Information Request by sending a Device Information Response. The required function is:  
`eCLD_ZIICommissionCommandDeviceInfoRspCommandSend()`
- 5. Identify Request (Optional):** The initiator may send an Identify Request to the node which has been chosen as the first Router of the new network, in order to confirm that the correct physical node is being commissioned. The required function is:  
`eCLD_ZIICommissionCommandDeviceIdentifyReqCommandSend()`
- 6. Network Start Request:** The initiator sends a Network Start Request to the chosen Router in order to create and start the network. The required function is:  
`eCLD_ZIICommissionCommandNetworkStartReqCommandSend()`
- 7. Network Start Response:** The Router replies to the Network Start Request by sending a Network Start Response. The required function is:  
`eCLD_ZIICommissionCommandNetworkStartRspCommandSend()`

Once the Router has started the network, the initiator joins the network (Router). The initiator then collects endpoint and cluster information from the Lighting device(s) on the Router node, and stores this information in a local lighting database.

Once the network (consisting of the initiator and one Router) is up and running, further nodes may be added as described in [Section 6.4.2](#).



**Figure 6: Creating a ZLL Network**

## 6.4.2 Adding to an Existing Network

A ZLL network (which has been set up as described in [Section 6.4.1](#)) can be extended by adding a node. The Touchlink extension process is described below and illustrated in [Figure 7](#) (also refer to the command list in [Table 27 on page 69](#)).



**Note:** Received Touchlink requests and responses are handled as ZigBee PRO events. The event handling is not detailed below but is outlined in [Section 6.6](#).

- 1. Scan Request:** The initiator sends a Scan Request to nodes in its vicinity. The required function is:  
`eCLD_ZIICommissionCommandScanReqCommandSend()`
- 2. Scan Response:** A receiving ZLL node replies to the Scan Request by sending a Scan Response. The required function is:  
`eCLD_ZIICommissionCommandScanRspCommandSend()`
- 3. Device Information Request:** The initiator sends a Device Information Request to those detected nodes that are of interest. The required function is:  
`eCLD_ZIICommissionCommandDeviceInfoReqCommandSend()`
- 4. Device Information Response:** A receiving node replies to the Device Information Request by sending a Device Information Response. The required function is:  
`eCLD_ZIICommissionCommandDeviceInfoRspCommandSend()`
- 5. Identify Request (Optional):** The initiator may send an Identify Request to the node which has been chosen to be added to the network, in order to confirm that the correct physical node is being commissioned. The required function is:  
`eCLD_ZIICommissionCommandDeviceIdentifyReqCommandSend()`
- 6. Network Join Request:** Depending on the target node type, the initiator sends a Network Join Router Request or Network Join End Device Request, as appropriate, to the target node. The required function is one of:  
`eCLD_ZIICommissionCommandNetworkJoinRouterReqCommandSend()`  
`eCLD_ZIICommissionCommandNetworkJoinEndDeviceReqCommandSend()`
- 7. Network Join Response:** Depending on the receiving node type, the node replies to the join request by sending a Network Join Router Response or Network Join End Device Response. The required function is one of:  
`eCLD_ZIICommissionCommandNetworkJoinRouterRspCommandSend()`  
`eCLD_ZIICommissionCommandNetworkJoinEndDeviceRspCommandSend()`

The node should now be a member of the network. The initiator then collects endpoint and cluster information from any Lighting device(s) on the new node, and stores this information in its local lighting database.

If the new node is to be used to control the light nodes of the network then it will need to learn certain information (such as controlled endpoints and configured groups) from the initiator - this is done using the Commissioning Utility commands, as described in [Section 6.5](#).

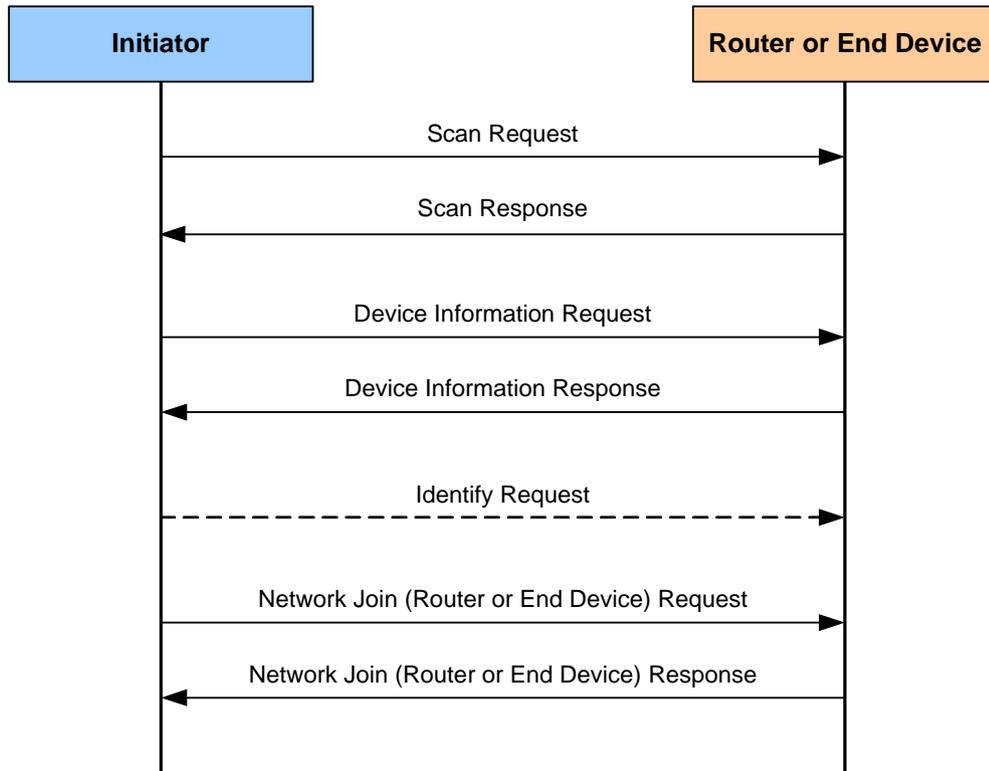


Figure 7: Extending a ZLL Network (Adding a Node)

### 6.4.3 Updating Network Settings

If one or more of the network settings change (e.g. the radio channel used), all nodes of the network need to be updated with the new settings.

To allow nodes to keep track of the status of the network settings, the Network Update Identifier is used. This identifier takes a value in the range 0x00 to 0xFF and is incremented when a network update has occurred (the value wraps around at 0xFF).

A node can be instructed to update its network settings by sending a Network Update Request to it. The required function is:

**eCLD\_ZllCommissionCommandNetworkUpdateReqCommandSend()**

The payload of the sent command contains the latest network settings and the current value of the Network Update Identifier (see [Section 6.8.17](#)). If the payload value is more recent than the value held by the target node, the node should update its network settings with those in the payload.

## 6.4.4 Stealing a Node

A node that is already part of a ZLL network can be taken or 'stolen' by another ZLL network using Touchlink (in which case, the stolen node will cease to be a member of its previous network). This transfer can only be performed on a node which supports one or more Lighting devices (and not Controller devices).

The node is stolen using an initiator in the new network, e.g. from a remote control unit. The 'stealing' process is as follows:

1. The initiator sends a Scan Request to nodes in its vicinity. The required function is:  
**eCLD\_ZIICommissionCommandScanReqCommandSend()**
2. A receiving ZLL node replies to the Scan Request by sending a Scan Response. The required function is:  
**eCLD\_ZIICommissionCommandScanRspCommandSend()**
3. The initiator receives Scan Responses from one or more nodes and, based on these responses, selects a node (containing a Lighting device) that is already a member of another ZLL network.
4. The initiator then sends a Reset To Factory New Request to the desired node. The required function is:  
**eCLD\_ZIICommissionCommandFactoryResetReqCommandSend()**
5. On receiving this request on the target node, the event `E_CLD_COMMISSION_CMD_FACTORY_RESET_REQ` is generated and the function **ZPS\_eApiZdoLeaveNetwork()** should be called. In addition, all persistent data should be reset.
6. The node can then be commissioned into the new network by following the process in [Section 6.4.2](#) from Step3.

Alternatively, instead of following the above process, a node can be stolen by either:

- Following the full process for creating a network in [Section 6.4.1](#) and calling **ZPS\_eApiZdoLeaveNetwork()** on the target node when a Network Start Request is received.
- Following the full process for adding a node in [Section 6.4.2](#) and calling **ZPS\_eApiZdoLeaveNetwork()** on the target node when a Network Join Router Request or Network Join End Device Request is received.



**Note:** If a node containing a Controller device (e.g. a remote control unit) is to be used in another ZLL network, it must first be reset using a Reset To Factory New Request. It can then used to create a new network (see [Section 6.4.1](#)) or to learn the control information of an existing network (see [Section 6.5](#)).

## 6.5 Using the Commissioning Utility

The Commissioning Utility is used when a ZLL network node needs to learn lighting control information (such as controlled endpoints and configured groups) from another node in the network. It is typically used when a new remote control unit is introduced into the network and needs to learn information from an existing remote control unit.

Unlike Touchlink, the Commissioning Utility can be incorporated in the main ZLL application on the node (and therefore use the same endpoint). This requires:

- a ZLL Commissioning cluster instance as a client to be created on the endpoint on the 'learner' node
- a ZLL Commissioning cluster instance as a server to be created on the endpoint on the 'teacher' node

A ZLL Commissioning cluster instance for the Commissioning Utility can be created using the function `eCLD_ZIIUtilityCreateUtility()`, on both nodes.

It is the responsibility of the learner node to request the required information from the teacher node. The Commissioning Utility command set is summarised in [Table 28](#). Commissioning Utility functions for issuing these commands are provided in the ZLL API and detailed in [Section 6.7.2](#).

Command	Identifier	Description
Endpoint information	0x40	Sends information about local endpoint (from teacher to learner)
Get Group Identifiers Request	0x41	Requests Group information from a remote node (from learner to teacher)
Get Endpoint List Request	0x42	Requests endpoint information from a remote node (from learner to teacher)

**Table 28: Commissioning Utility Commands**

Use of the above commands and associated functions is described below and is illustrated in [Figure 8](#).



**Note:** Received Commissioning Utility requests and responses are handled as ZigBee PRO events by the ZLL profile library (this event handling is therefore transparent to the application).

1. **Endpoint Information command:** The teacher node first sends an Endpoint Information command containing basic information about its local endpoint (IEEE address, network address endpoint number, Profile ID, Device ID) to the learner node. The required function is:

**eCLD\_ZIIUtilityCommandEndpointInformationCommandSend()**

Note that the teacher node will already have the relevant target endpoint on the learner node from the joining process (described in [Section 6.4](#)).

2. **Get Endpoint List Request:** The learner node then sends a Get Endpoint List Request to the teacher node to request information about the remote endpoints that the teacher node controls. The required function is:

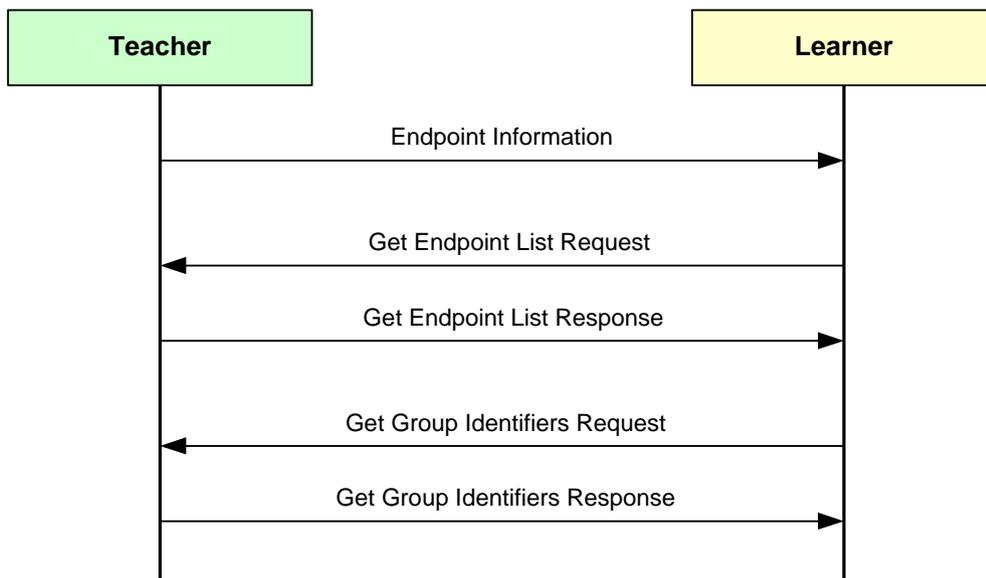
**eCLD\_ZIIUtilityCommandGetEndpointListReqCommandSend()**

The teacher node automatically replies to the Get Endpoint List Request by sending a Get Endpoint List Response containing the requested information.

3. **Get Group Identifiers Request:** The learner node then sends a Get Group Identifiers Request to the teacher node to request a list of the lighting groups configured on the teacher node. The required function is:

**eCLD\_ZIIUtilityCommandGetGroupIdReqCommandSend()**

The teacher node automatically replies to the Get Group Identifiers Request by sending a Get Group Identifiers Response containing the requested information.



**Figure 8: Learning Process**

To complete the learning process, the learner node may need other information which can be acquired using commands/functions of the relevant cluster (described in the *ZCL User Guide (JN-UG-3103)*).

---

## 6.6 ZLL Commissioning Events (Touchlink)

ZLL Commissioning cluster events that result from receiving Touchlink requests and responses must be handled at the application level (while events that result from Commissioning Utility requests and responses are handled by the ZLL profile library).

When a Touchlink request or response command (e.g. a Scan Request) is received by a node, a stack event is generated which is wrapped in a `tsZCL_CallbackEvent` structure by the ZLL profile. In this structure:

- `eEventType` field is set to `E_ZCL_CBET_CLUSTER_CUSTOM`
- `sClusterCustomMessage` field's `tsZCL_ClusterCustomMessage` structure is filled in by:
  - setting `u16ClusterId` to `ZLL_CLUSTER_ID_COMMISSIONING`
  - pointing `pvCustomData` to the payload data of the received command

For details of the above structures, refer to the *ZCL User Guide (JN-UG-3103)*.

The payload data contains a command ID, which uses one of the enumerations listed in [Section 6.6.1](#). The ZLL profile passes the event to the ZCL event handler to check that the command ID is valid for the target endpoint. If it is valid, the ZLL profile invokes the user-defined callback function that was specified through the function **eZLL\_RegisterCommissionEndPoint()**. The callback function can access the payload through the `tsCLD_ZllCommissionCustomDataStructure` structure, which is created when the above function is called.

Thus, the above user-defined callback function must be designed to handle the relevant Touchlink events:

- For a request, the callback function may need to populate a structure with the required data and send a response using the appropriate response function, e.g. by calling **eCLD\_ZllCommissionCommandScanRspCommandSend()** to respond to a Scan Request.
- For a response, the callback function may just need to extract the returned data from the event.

Alternatively, the callback function may simply notify the main application of the received command and provide the payload, so that the application can process the command.

The handling of Touchlink events is illustrated in [Figure 9](#).

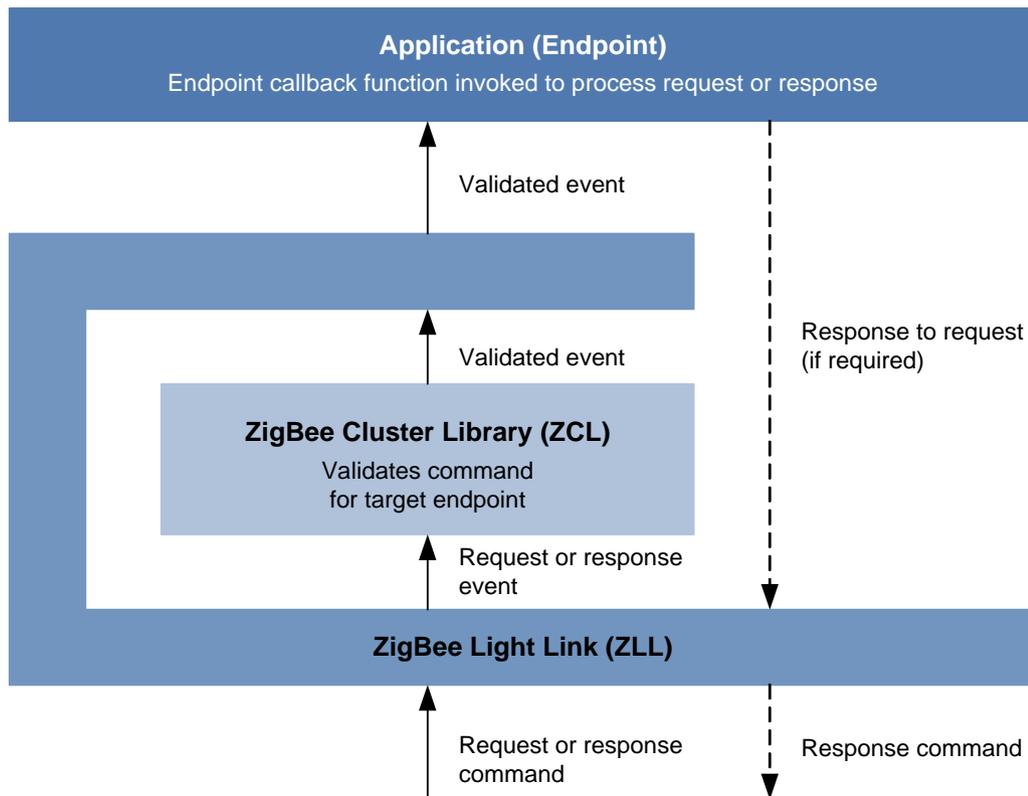


Figure 9: Touchlink Event Handling

## 6.6.1 Touchlink Command Events

The events that can be generated for Touchlink are listed and described below (the enumerations are defined in the structure `teCLD_ZllCommission_Command`, shown in [Section 6.9.1](#)).

Event	Description
E_CLD_COMMISSION_CMD_SCAN_REQ	A Scan Request has been received (by server)
E_CLD_COMMISSION_CMD_SCAN_RSP	A Scan Response has been received (by client)
E_CLD_COMMISSION_CMD_DEVICE_INFO_REQ	A Device Information Request has been received (by server)
E_CLD_COMMISSION_CMD_DEVICE_INFO_RSP	A Device Information Response has been received (by client)
E_CLD_COMMISSION_CMD_IDENTIFY_REQ	An Identify Request has been received (by server)
E_CLD_COMMISSION_CMD_FACTORY_RESET_REQ	A Reset To Factory New Request has been received (by server)
E_CLD_COMMISSION_CMD_NETWORK_START_REQ	A Network Start Request has been received (by server)
E_CLD_COMMISSION_CMD_NETWORK_START_RSP	A Network Start Response has been received (by client)
E_CLD_COMMISSION_CMD_NETWORK_JOIN_ROUTER_REQ	A Network Join Router Request has been received (by server)
E_CLD_COMMISSION_CMD_NETWORK_JOIN_ROUTER_RSP	A Network Join Router Response has been received (by client)
E_CLD_COMMISSION_CMD_NETWORK_JOIN_END_DEVICE_REQ	A Network Join End Device Request has been received (by server)
E_CLD_COMMISSION_CMD_NETWORK_JOIN_END_DEVICE_RSP	A Network Join End Device Response has been received (by client)
E_CLD_COMMISSION_CMD_NETWORK_UPDATE_REQ	A Network Update Request has been received (by server)

**Table 29: Touchlink Events**

---

## 6.6.2 Commissioning Utility Command Events

The events that can be generated for the Commissioning Utility are listed and described below (the enumerations are defined in the structure `teCLD_ZllUtility_Command`, shown in [Section 6.9.2](#)).

Event	Description
E_CLD_UTILITY_CMD_ENDPOINT_INFO	An Endpoint Information command has been received (by client)
E_CLD_UTILITY_CMD_GET_GROUP_ID_REQ_RSP	A Get Group Identifiers Request has been received (by server) or a Get Group Identifiers Response has been received (by client)
E_CLD_UTILITY_CMD_GET_ENDPOINT_LIST_REQ_RSP	A Get Endpoint List Request has been received (by server) or a Get Endpoint List Response has been received (by client)

**Table 30: Touchlink Events**

---

## 6.7 Functions

The functions of the ZLL Commissioning cluster are divided into two categories:

- Touchlink functions, detailed in [Section 6.7.1](#)
- Commissioning Utility functions, detailed in [Section 6.7.2](#)

---

## 6.7.1 Touchlink Functions

The following Touchlink functions are provided in the ZLL API:

<b>Function</b>	<b>Page</b>
<a href="#">eZLL_RegisterCommissionEndPoint</a>	82
<a href="#">eCLD_ZIICommissionCreateCommission</a>	83
<a href="#">eCLD_ZIICommissionCommandScanReqCommandSend</a>	84
<a href="#">eCLD_ZIICommissionCommandScanRspCommandSend</a>	85
<a href="#">eCLD_ZIICommissionCommandDeviceInfoReqCommandSend</a>	86
<a href="#">eCLD_ZIICommissionCommandDeviceInfoRspCommandSend</a>	87
<a href="#">eCLD_ZIICommissionCommandDeviceIdentifyReqCommandSend</a>	88
<a href="#">eCLD_ZIICommissionCommandFactoryResetReqCommandSend</a>	89
<a href="#">eCLD_ZIICommissionCommandNetworkStartReqCommandSend</a>	90
<a href="#">eCLD_ZIICommissionCommandNetworkStartRspCommandSend</a>	91
<a href="#">eCLD_ZIICommissionCommandNetworkJoinRouterReqCommandSend</a>	92
<a href="#">eCLD_ZIICommissionCommandNetworkJoinRouterRspCommandSend</a>	93
<a href="#">eCLD_ZIICommissionCommandNetworkJoinEndDeviceReqCommandSend</a>	94
<a href="#">eCLD_ZIICommissionCommandNetworkJoinEndDeviceRspCommandSend</a>	95
<a href="#">eCLD_ZIICommissionCommandNetworkUpdateReqCommandSend</a>	96

---

## eZLL\_RegisterCommissionEndPoint

---

```
teZCL_Status eZLL_RegisterCommissionEndPoint(  
    uint8 u8EndPointIdentifier,  
    tfpZCL_ZCLCallbackFunction cbCallBack,  
    tsZLL_CommissionEndPoint *psDeviceInfo);
```

### Description

This function registers a 'commissioning' endpoint for Touchlink and creates a ZLL Commissioning cluster instance on the endpoint.

Touchlink must have its own application (separate from the main ZLL application) on its own endpoint.

This function uses **eCLD\_ZICommissionCreateCommission()** to create the cluster instance. The type of cluster instance to be created (server or client, or both) is determined using the compile-time options in the header file **zcl\_options.h** (refer to [Section 6.10](#)).

### Parameters

<i>u8EndPointIdentifier</i>	Identifier of endpoint to be registered - this is an endpoint number in the range 1 to 240
<i>cbCallBack</i>	Pointer to a callback function to handle events associated with the registered endpoint
<i>psDeviceInfo</i>	Pointer to structure to be used to hold Touchlink endpoint information (see <a href="#">Section 6.8.1</a> )

### Returns

E\_ZCL\_SUCCESS

---

## eCLD\_ZIICommissionCreateCommission

---

```

teZCL_Status eCLD_ZIICommissionCreateCommission(
    tsZCL_ClusterInstance *psClusterInstance,
    bool_t blsServer,
    tsZCL_ClusterDefinition *psClusterDefinition,
    void *pvSharedStructPtr,
    tsZCL_AttributeStatus *psAttributeStatus,
    tsCLD_ZIICommissionCustomDataStructure
        *psCustomDataStructure);
    
```

### Description

This function creates a ZLL Commissioning cluster instance for Touchlink on the endpoint of the calling application. The type of cluster instance (server or client) to be created must be specified.

In practice, this function does not need to be called explicitly by the application, as the function **eZLL\_RegisterCommissionEndPoint()** calls this function to create the cluster instance.

### Parameters

<i>psClusterInstance</i>	Pointer to cluster instance structure on local endpoint
<i>blsServer</i>	Type of cluster instance (server or client) to be created: TRUE - server FALSE - client
<i>psClusterDefinition</i>	Pointer to cluster definition structure containing information about the cluster
<i>pvSharedStructPtr</i>	Pointer to structure containing the shared storage for the cluster
<i>psAttributeStatus</i>	Pointer to a structure containing the storage for each attribute's status
<i>psCustomDataStructure</i>	Pointer to custom data to be provided to the cluster (see <a href="#">Section 6.8.3</a> )

### Returns

E\_ZCL\_SUCCESS

---

## eCLD\_ZIICommissionCommandScanReqCommandSend

---

```
teZCL_Status  
eCLD_ZIICommissionCommandScanReqCommandSend(  
    ZPS_tsInterPanAddress *psDestinationAddress,  
    uint8 *pu8TransactionSequenceNumber,  
    tsCLD_ZIICommission_ScanReqCommandPayload  
        *psPayload);
```

### Description

This function is used to send a Scan Request command to initiate a scan for other nodes in the local neighbourhood. The command is sent as an inter-PAN message.

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

### Parameters

<i>psDestinationAddress</i>	Pointer to structure containing PAN ID and address information for target node(s)
<i>pu8TransactionSequenceNumber</i>	Pointer to a location to store the Transaction Sequence Number (TSN) of the request
<i>psPayload</i>	Pointer to structure containing payload data for the Scan Request command (see <a href="#">Section 6.8.5</a> )

### Returns

E\_ZCL\_SUCCESS

---

## eCLD\_ZIICommissionCommandScanRspCommandSend

---

```
PUBLIC teZCL_Status  
eCLD_ZIICommissionCommandScanRspCommandSend(  
    ZPS_tsInterPanAddress *psDestinationAddress,  
    uint8 *pu8TransactionSequenceNumber,  
    tsCLD_ZIICommission_ScanRspCommandPayload  
        *psPayload);
```

### Description

This function is used to send a Scan Response command containing information about the local node in reply to a received Scan Request from a remote node. The command is sent as an inter-PAN message.

A pointer must be provided to a structure containing the data to be returned.

The specified Transaction Sequence Number (TSN) of the response must match the TSN of the corresponding request, as this will allow the response to be paired with the request at the destination.

### Parameters

<i>psDestinationAddress</i>	Pointer to structure containing PAN ID and address information for target node
<i>pu8TransactionSequenceNumber</i>	Pointer to location containing the Transaction Sequence Number (TSN) of the response
<i>psPayload</i>	Pointer to structure containing payload data for the Scan Response command (see <a href="#">Section 6.8.6</a> )

### Returns

E\_ZCL\_SUCCESS

---

## eCLD\_ZIICommissionCommandDeviceInfoReqCommandSend

---

```
teZCL_Status  
eCLD_ZIICommissionCommandDeviceInfoReqCommandSend(  
ZPS_tsInterPanAddress *psDestinationAddress,  
uint8 *pu8TransactionSequenceNumber,  
tsCLD_ZIICommission_DeviceInfoReqCommandPayload  
*psPayload);
```

### Description

This function is used to send a Device Information Request command to obtain information about the devices on a remote node. The command is sent as an inter-PAN message.

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

### Parameters

<i>psDestinationAddress</i>	Pointer to structure containing PAN ID and address information for target node
<i>pu8TransactionSequenceNumber</i>	Pointer to a location to store the Transaction Sequence Number (TSN) of the request
<i>psPayload</i>	Pointer to structure containing payload data for the Device Information Request command (see <a href="#">Section 6.8.7</a> )

### Returns

E\_ZCL\_SUCCESS

---

**eCLD\_ZIICommissionCommandDeviceInfoRspCommandSend**

---

```
PUBLIC teZCL_Status  
eCLD_ZIICommissionCommandDeviceInfoRspCommandSend(  
    ZPS_tsInterPanAddress *psDestinationAddress,  
    uint8 *pu8TransactionSequenceNumber,  
    tsCLD_ZIICommission_DeviceInfoRspCommandPayload  
        *psPayload);
```

**Description**

This function is used to send a Device Information Response command containing information about the devices on the local node in reply to a received Device Information Request from a remote node. The command is sent as an inter-PAN message.

A pointer must be provided to a structure containing the data to be returned.

The specified Transaction Sequence Number (TSN) of the response must match the TSN of the corresponding request, as this will allow the response to be paired with the request at the destination.

**Parameters**

<i>psDestinationAddress</i>	Pointer to structure containing PAN ID and address information for target node
<i>pu8TransactionSequenceNumber</i>	Pointer to location containing the Transaction Sequence Number (TSN) of the response
<i>psPayload</i>	Pointer to structure containing payload data for the Device Information Response command (see <a href="#">Section 6.8.8</a> )

**Returns**

E\_ZCL\_SUCCESS

---

## eCLD\_ZllCommissionCommandDeviceIdentifyReqCommandSend

---

```
teZCL_Status  
eCLD_ZllCommissionCommandDeviceIdentifyReqCommandSend(  
    ZPS_tsInterPanAddress *psDestinationAddress,  
    uint8 *pu8TransactionSequenceNumber,  
    tsCLD_ZllCommission_IdentifyReqCommandPayload  
        *psPayload);
```

### Description

This function is used to send an Identify Request command to ask a remote node to identify itself by entering 'identify mode' (this is a visual indication, such as flashing a LED). The command is sent as an inter-PAN message.

The command payload contains a value indicating the length of time, in seconds, that the target device should remain in identify mode. It is also possible to use this command to instruct the target node to immediately exit identify mode (if it is already in this mode).

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

### Parameters

<i>psDestinationAddress</i>	Pointer to structure containing PAN ID and address information for target node
<i>pu8TransactionSequenceNumber</i>	Pointer to a location to store the Transaction Sequence Number (TSN) of the request
<i>psPayload</i>	Pointer to structure containing payload data for the Identify Request command (see <a href="#">Section 6.8.9</a> )

### Returns

E\_ZCL\_SUCCESS

---

## eCLD\_ZIICommissionCommandFactoryResetReqCommandSend

---

```

teZCL_Status
eCLD_ZIICommissionCommandFactoryResetReqCommandSend(
    ZPS_tsInterPanAddress *psDestinationAddress,
    uint8 *pu8TransactionSequenceNumber,
    tsCLD_ZIICommission_FactoryResetReqCommandPayload
    *psPayload);

```

### Description

This function is used to send a Reset to Factory New Request command to ask a remote node to return to its 'factory new' state. The command is sent as an inter-PAN message.

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

### Parameters

<i>psDestinationAddress</i>	Pointer to structure containing PAN ID and address information for target node
<i>pu8TransactionSequenceNumber</i>	Pointer to a location to store the Transaction Sequence Number (TSN) of the request
<i>psPayload</i>	Pointer to structure containing payload data for the Reset to Factory New Request command (see <a href="#">Section 6.8.10</a> )

### Returns

E\_ZCL\_SUCCESS

---

## eCLD\_ZIICommissionCommandNetworkStartReqCommandSend

---

```
teZCL_Status  
eCLD_ZIICommissionCommandNetworkStartReqCommandSend(  
    ZPS_tsInterPanAddress *psDestinationAddress,  
    uint8 *pu8TransactionSequenceNumber,  
    tsCLD_ZIICommission_NetworkStartReqCommandPayload  
        *psPayload);
```

### Description

This function is used to send a Network Start Request command to create a new network with a detected Router. The command is sent as an inter-PAN message.

The function is called once the results of a Scan Request command have been received and a detected Router has been selected.

The command payload contains information about the network and the local node, as well as certain data for the target node. This payload information is detailed in [Section 6.8.11](#).

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

### Parameters

<i>psDestinationAddress</i>	Pointer to structure containing PAN ID and address information for target node
<i>pu8TransactionSequenceNumber</i>	Pointer to a location to store the Transaction Sequence Number (TSN) of the request
<i>psPayload</i>	Pointer to structure containing payload data for the Network Start Request command (see <a href="#">Section 6.8.11</a> )

### Returns

E\_ZCL\_SUCCESS

---

**eCLD\_ZIICommissionCommandNetworkStartRspCommandSend**

---

```
PUBLIC teZCL_Status  
eCLD_ZIICommissionCommandNetworkStartRspCommandSend(  
    ZPS_tsInterPanAddress *psDestinationAddress,  
    uint8 *pu8TransactionSequenceNumber,  
    tsCLD_ZIICommission_NetworkStartRspCommandPayload  
        *psPayload);
```

**Description**

This function is used to send a Network Start Response command to confirm that the local (Router) node is ready to be the first node to join a newly created network in reply to a received Network Start Request from a remote node. The command is sent as an inter-PAN message.

A pointer must be provided to a structure containing the data to be returned.

The specified Transaction Sequence Number (TSN) of the response must match the TSN of the corresponding request, as this will allow the response to be paired with the request at the destination.

**Parameters**

<i>psDestinationAddress</i>	Pointer to structure containing PAN ID and address information for target node
<i>pu8TransactionSequenceNumber</i>	Pointer to location containing the Transaction Sequence Number (TSN) of the response
<i>psPayload</i>	Pointer to structure containing payload data for the Network Start Response command (see <a href="#">Section 6.8.12</a> )

**Returns**

E\_ZCL\_SUCCESS

---

## eCLD\_ZIICommissionCommandNetworkJoinRouterReqCommandSend

---

```
teZCL_Status  
eCLD_ZIICommissionCommandNetworkJoinRouterReqCommandSend(  
    ZPS_tsInterPanAddress *psDestinationAddress,  
    uint8 *pu8TransactionSequenceNumber,  
    tsCLD_ZIICommission_NetworkJoinRouterReqCommandPayload  
        *psPayload);
```

### Description

This function is used to send a Network Join Router Request command to allow a detected Router to join the created network. The command is sent as an inter-PAN message.

The function can be called once a network has been created. The target Router is distinct from the Router that was included when network was created.

The command payload contains information about the network and the local node, as well as certain data for the target node. This payload information is detailed in [Section 6.8.13](#).

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

### Parameters

<i>psDestinationAddress</i>	Pointer to structure containing PAN ID and address information for target node
<i>pu8TransactionSequenceNumber</i>	Pointer to a location to store the Transaction Sequence Number (TSN) of the request
<i>psPayload</i>	Pointer to structure containing payload data for the Network Join Router Request command (see <a href="#">Section 6.8.13</a> )

### Returns

E\_ZCL\_SUCCESS

---

## eCLD\_ZIICommissionCommandNetworkJoinRouterRspCommandSend

---

```

PUBLIC teZCL_Status
eCLD_ZIICommissionCommandNetworkJoinRouterRspCommandSend(
    ZPS_tsInterPanAddress psDestinationAddress,
    uint8 *pu8TransactionSequenceNumber,
    tsCLD_ZIICommission_NetworkJoinRouterRspCommandPayload
        *psPayload);
    
```

### Description

This function is used to send a Network Join Router Response command to confirm that the local (Router) node is ready to join a network in reply to a received Network Join Router Request from a remote node. The command is sent as an inter-PAN message.

A pointer must be provided to a structure containing the data to be returned.

The specified Transaction Sequence Number (TSN) of the response must match the TSN of the corresponding request, as this will allow the response to be paired with the request at the destination.

### Parameters

<i>psDestinationAddress</i>	Pointer to structure containing PAN ID and address information for target node
<i>pu8TransactionSequenceNumber</i>	Pointer to location containing the Transaction Sequence Number (TSN) of the response
<i>psPayload</i>	Pointer to structure containing payload data for the Network Join Router Response command (see <a href="#">Section 6.8.14</a> )

### Returns

E\_ZCL\_SUCCESS

---

## eCLD\_ZIICommissionCommandNetworkJoinEndDeviceReqCommandSend

---

```
teZCL_Status  
eCLD_ZIICommissionCommandNetworkJoinEndDeviceReqCommandSend(  
    ZPS_tsInterPanAddress *psDestinationAddress,  
    uint8 *pu8TransactionSequenceNumber,  
    tsCLD_ZIICommission_NetworkJoinEndDeviceReqCommandPayload  
        *psPayload);
```

### Description

This function is used to send a Network Join End Device Request command to allow a detected End Device to join the created network. The command is sent as an inter-PAN message.

The function can be called once a network has been created.

The command payload contains information about the network and the local node, as well as certain data for the target node. This data includes a range of network addresses and a range of group IDs from which the target End Device can assign values to the other nodes - in this case, the End Device would typically be a remote control unit. This payload information is detailed in [Section 6.8.15](#).

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

### Parameters

<i>psDestinationAddress</i>	Pointer to structure containing PAN ID and address information for target node
<i>pu8TransactionSequenceNumber</i>	Pointer to a location to store the Transaction Sequence Number (TSN) of the request
<i>psPayload</i>	Pointer to structure containing payload data for the Network Join End Device Request command (see <a href="#">Section 6.8.15</a> )

### Returns

E\_ZCL\_SUCCESS

---

## eCLD\_ZIICommissionCommandNetworkJoinEndDeviceRspCommandSend

---

```

PUBLIC teZCL_Status
eCLD_ZIICommissionCommandNetworkJoinEndDeviceRspCommandSend(
    ZPS_tsInterPanAddress *psDestinationAddress,
    uint8 *pu8TransactionSequenceNumber,
    tsCLD_ZIICommission_NetworkJoinEndDeviceRspCommandPayload
        *psPayload);
    
```

### Description

This function is used to send a Network Join End Device Response command to confirm that the local (End Device) node is ready to join a network in reply to a received Network Join End Device Request from a remote node. The command is sent as an inter-PAN message.

A pointer must be provided to a structure containing the data to be returned.

The specified Transaction Sequence Number (TSN) of the response must match the TSN of the corresponding request, as this will allow the response to be paired with the request at the destination.

### Parameters

<i>psDestinationAddress</i>	Pointer to structure containing PAN ID and address information for target node
<i>pu8TransactionSequenceNumber</i>	Pointer to location containing the Transaction Sequence Number (TSN) of the response
<i>psPayload</i>	Pointer to structure containing payload data for the Network Join End Device Response command (see <a href="#">Section 6.8.16</a> )

### Returns

E\_ZCL\_SUCCESS

---

## eCLD\_ZIICommissionCommandNetworkUpdateReqCommandSend

---

```
teZCL_Status  
eCLD_ZIICommissionCommandNetworkUpdateReqCommandSend(  
    ZPS_tsInterPanAddress *psDestinationAddress,  
    uint8 *pu8TransactionSequenceNumber,  
    tsCLD_ZIICommission_NetworkUpdateReqCommandPayload  
        *psPayload);
```

### Description

This function is used to send a Network Update Request command to bring a node that has missed a network update back into the network. The command is sent as an inter-PAN message.

The command payload contains information about the network, including the current value of the Network Update Identifier. This identifier takes a value in the range 0x00 to 0xFF and is incremented when a network update has occurred (the value wraps around at 0xFF). Thus, if this value in the payload is more recent than the value of this identifier held by the target node, the node should update its network settings using the values in the rest of the payload. The payload information is detailed in [Section 6.8.17](#).

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

### Parameters

<i>psDestinationAddress</i>	Pointer to structure containing PAN ID and address information for target node
<i>pu8TransactionSequenceNumber</i>	Pointer to a location to store the Transaction Sequence Number (TSN) of the request
<i>psPayload</i>	Pointer to structure containing payload data for the Network Update Request command (see <a href="#">Section 6.8.17</a> )

### Returns

E\_ZCL\_SUCCESS

---

## 6.7.2 Commissioning Utility Functions

The following Commissioning Utility functions are provided in the ZLL API:

<b>Function</b>	<b>Page</b>
<a href="#">eCLD_ZIIUtilityCreateUtility</a>	98
<a href="#">eCLD_ZIIUtilityCommandEndpointInformationCommandSend</a>	99
<a href="#">eCLD_ZIIUtilityCommandGetGroupIdReqCommandSend</a>	100
<a href="#">eCLD_ZIIUtilityCommandGetGroupIdRspCommandSend</a>	101
<a href="#">eCLD_ZIIUtilityCommandGetEndpointListReqCommandSend</a>	102
<a href="#">eCLD_ZIIUtilityCommandGetEndpointListRspCommandSend</a>	103
<a href="#">eCLD_ZIIUtilityCommandHandler</a>	104

---

## eCLD\_ZIIUtilityCreateUtility

---

```
teZCL_Status eCLD_ZIIUtilityCreateUtility(  
    tsZCL_ClusterInstance *psClusterInstance,  
    bool_t blsServer,  
    tsZCL_ClusterDefinition *psClusterDefinition,  
    void *pvSharedStructPtr,  
    tsZCL_AttributeStatus psAttributeStatus,  
    tsCLD_ZIIUtilityCustomDataStructure  
        *psCustomDataStructure);
```

### Description

This function creates a ZLL Commissioning cluster instance for the Commissioning Utility. The cluster instance is created on the endpoint of the calling application, which should be the main ZLL application on the node. The type of cluster instance (server or client) to be created must be specified.

### Parameters

<i>psClusterInstance</i>	Pointer to cluster instance structure on local endpoint
<i>blsServer</i>	Type of cluster instance (server or client) to be created: TRUE - server FALSE - client
<i>psClusterDefinition</i>	Pointer to cluster definition structure containing information about the cluster
<i>pvSharedStructPtr</i>	Pointer to structure containing the shared storage for the cluster
<i>psAttributeStatus</i>	Pointer to a structure containing the storage for each attribute's status
<i>psCustomDataStructure</i>	Pointer to custom data to be provided to the cluster (see <a href="#">Section 6.8.20</a> )

### Returns

E\_ZCL\_SUCCESS

---

## eCLD\_ZIIUtilityCommandEndpointInformationCommandSend

---

```
teZCL_Status  
eCLD_ZIIUtilityCommandEndpointInformationCommandSend(  
    uint8 u8SrcEndpoint,  
    uint8 u8DstEndpoint,  
    tsZCL_Address *psDestinationAddress,  
    uint8 *pu8TransactionSequenceNumber,  
    tsCLD_ZIIUtility_EndpointInformationCommandPayload  
        *psPayload);
```

### Description

This function is used to send an Endpoint Information command to provide a remote endpoint with general information about the local endpoint (this may prompt the remote endpoint to request further information about the local endpoint). The function would typically be used to send local endpoint information from a 'teacher' node to a 'learner' node, in order to facilitate two-way communication between the Commissioning Utilities on the two nodes.

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

### Parameters

<i>u8SrcEndpoint</i>	Number of local endpoint (1-240)
<i>u8DstEndpoint</i>	Number of destination endpoint (1-240)
<i>psDestinationAddress</i>	Pointer to structure containing address information for target node
<i>pu8TransactionSequenceNumber</i>	Pointer to a location to store the Transaction Sequence Number (TSN) of the command
<i>psPayload</i>	Pointer to structure to contain payload data for the Endpoint Information command (see <a href="#">Section 6.8.20</a> )

### Returns

E\_ZCL\_SUCCESS

---

## eCLD\_ZIIUtilityCommandGetGroupIdReqCommandSend

---

```
teZCL_Status  
eCLD_ZIIUtilityCommandGetGroupIdReqCommandSend(  
    uint8 u8SrcEndpoint,  
    uint8 u8DstEndpoint,  
    tsZCL_Address *psDestinationAddress,  
    uint8 *pu8TransactionSequenceNumber,  
    uint8 u8StartIndex);
```

### Description

This function is used to send a Get Group Identifiers Request command to obtain information about the groups (of lights) that have been configured on a remote endpoint. The function would typically be used on a 'learner' node to request the groups that have been configured on a 'teacher' node.

The first group from the groups list to be included in the returned information must be specified in terms of an index.

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

### Parameters

<i>u8SrcEndpoint</i>	Number of local endpoint (1-240)
<i>u8DstEndpoint</i>	Number of destination endpoint (1-240)
<i>psDestinationAddress</i>	Pointer to structure containing address information for target node
<i>pu8TransactionSequenceNumber</i>	Pointer to a location to store the Transaction Sequence Number (TSN) of the request
<i>u8StartIndex</i>	Index in group list of the first group to include in the returned information

### Returns

E\_ZCL\_SUCCESS

---

## eCLD\_ZIIUtilityCommandGetGroupIdRspCommandSend

---

```

PUBLIC teZCL_Status
eCLD_ZIIUtilityCommandGetGroupIdRspCommandSend(
    uint8 u8SrcEndpoint,
    uint8 u8DstEndpoint,
    tsZCL_Address *psDestinationAddress,
    uint8 *pu8TransactionSequenceNumber,
    uint8 u8StartIndex);
    
```

### Description

This function is used to send a Get Group Identifiers Response command containing information about the groups (of lights) that have been configured on the local endpoint. The function would typically be used on a 'teacher' node to respond to a Get Group Identifiers Request from a 'learner' node.

The first group from the groups list to be included in the returned information must be specified in terms of an index. The returned information includes this index, the number of (consecutive) groups included and the identifier of each group.

The specified Transaction Sequence Number (TSN) of the response must match the TSN of the corresponding request, as this will allow the response to be paired with the request at the destination.

### Parameters

<i>u8SrcEndpoint</i>	Number of local endpoint (1-240)
<i>u8DstEndpoint</i>	Number of destination endpoint (1-240)
<i>psDestinationAddress</i>	Pointer to stucture containing address information for target node
<i>pu8TransactionSequenceNumber</i>	Pointer to location containing the Transaction Sequence Number (TSN) of the response
<i>u8StartIndex</i>	Index in group list of the first group to include in the returned information

### Returns

E\_ZCL\_SUCCESS

---

## eCLD\_ZIIUtilityCommandGetEndpointListReqCommandSend

---

```
teZCL_Status  
eCLD_ZIIUtilityCommandGetEndpointListReqCommandSend(  
    uint8 u8SrcEndpoint,  
    uint8 u8DstEndpoint,  
    tsZCL_Address *psDestinationAddress,  
    uint8 *pu8TransactionSequenceNumber,  
    uint8 u8StartIndex);
```

### Description

This function is used to send a Get Endpoint List Request command to obtain information about controlled endpoints. The function would typically be used on a 'learner' node to request the remote endpoints that a 'teacher' node controls.

The first endpoint from the endpoints list to be included in the returned information must be specified in terms of an index.

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

### Parameters

<i>u8SrcEndpoint</i>	Number of local endpoint (1-240)
<i>u8DstEndpoint</i>	Number of destination endpoint (1-240)
<i>psDestinationAddress</i>	Pointer to structure containing address information for target node
<i>pu8TransactionSequenceNumber</i>	Pointer to a location to store the Transaction Sequence Number (TSN) of the request
<i>u8StartIndex</i>	Index in endpoint list of the first endpoint to include in the returned information

### Returns

E\_ZCL\_SUCCESS

---

## eCLD\_ZIIUtilityCommandGetEndpointListRspCommandSend

---

```

PUBLIC teZCL_Status
eCLD_ZIIUtilityCommandGetEndpointListRspCommandSend(
    uint8 u8SrcEndpoint,
    uint8 u8DstEndpoint,
    tsZCL_Address *psDestinationAddress,
    uint8 *pu8TransactionSequenceNumber,
    uint8 u8StartIndex);
    
```

### Description

This function is used to send a Get Endpoint List Response command containing information about controlled endpoints. The function would typically be used on a ‘teacher’ node to respond to a Get Endpoint List Request from a ‘learner’ node.

The first endpoint from the endpoints list to be included in the returned information must be specified in terms of an index. The returned information will include this index, the number of (consecutive) endpoints included and the information about each endpoint (including endpoint number, identifier of supported application profile, identifier of resident ZLL device and version of this device).

The specified Transaction Sequence Number (TSN) of the response must match the TSN of the corresponding request, as this will allow the response to be paired with the request at the destination.

### Parameters

<i>u8SrcEndpoint</i>	Number of local endpoint (1-240)
<i>u8DstEndpoint</i>	Number of destination endpoint (1-240)
<i>psDestinationAddress</i>	Pointer to stucture containing address information for target node
<i>pu8TransactionSequenceNumber</i>	Pointer to location containing the Transaction Sequence Number (TSN) of the response
<i>u8StartIndex</i>	Index in endpoint list of the first endpoint to include in the returned information

### Returns

E\_ZCL\_SUCCESS

---

## eCLD\_ZIIUtilityCommandHandler

---

```
teZCL_Status eCLD_ZIIUtilityCommandHandler(  
    ZPS_tsAfEvent *pZPSevent,  
    tsZCL_EndPointDefinition *psEndPointDefinition,  
    tsZCL_ClusterInstance *psClusterInstance);
```

### Description

This function parses a ZigBee PRO event and invokes the user-defined callback function that has been registered for the device (using the relevant endpoint registration function, described in [Chapter 7](#)).

The registered user-defined callback function must be designed to handle events associated with the Commissioning Utility.

### Parameters

<i>pZPSevent</i>	Pointer to received ZigBee PRO event
<i>psEndPointDefinition</i>	Pointer to structure which defines endpoint on which ZLL Commissioning cluster (utility) resides
<i>psClusterInstance</i>	Pointer to ZLL Commissioning cluster (utility) instance structure

### Returns

E\_ZCL\_SUCCESS

---

## 6.8 Structures

This section details the structures used in the ZLL Commissioning cluster (both Touchlink and Commissioning Utility parts).

---

### 6.8.1 tsZLL\_CommissionEndpoint

This structure is used to hold endpoint information for a Touchlink application.

```
typedef struct
{
    tsZCL_EndPointDefinition sEndPoint;
    tsZLL_CommissionEndpointClusterInstances sClusterInstance;

#ifdef CLD_ZLL_COMMISSION && ZLL_COMMISSION_SERVER
    tsCLD_ZllCommission sZllCommissionServerCluster;
    tsCLD_ZllCommissionCustomDataStructure
        sZllCommissionServerCustomDataStructure;
#endif

#ifdef CLD_ZLL_COMMISSION && ZLL_COMMISSION_CLIENT
    tsCLD_ZllCommission sZllCommissionClientCluster;
    tsCLD_ZllCommissionCustomDataStructure
        sZllCommissionClientCustomDataStructure;
#endif

} tsZLL_CommissionEndpoint;
```

where:

- `sEndPoint` is a ZCL structure containing information about the endpoint (refer to the *ZCL User Guide (JN-UG-3103)*).
- `sClusterInstance` is a structure containing information about the ZLL Commissioning cluster instance on the endpoint (see [Section 6.8.2](#)).
- For a Touchlink server, the following fields are used:
  - `sZllCommissionServerCluster` is the ZLL Commissioning cluster structure (which contains no attributes).
  - `sZllCommissionServerCustomDataStructure` is a structure containing custom data for the cluster server (see [Section 6.8.3](#)).
- For a Touchlink client, the following fields are used:
  - `sZllCommissionClientCluster` is the ZLL Commissioning cluster structure (which contains no attributes).
  - `sZllCommissionClientCustomDataStructure` is a structure containing custom data for the cluster client (see [Section 6.8.3](#)).

---

## 6.8.2 tsZLL\_CommissionEndpointClusterInstances

This structure holds information about the ZLL Commissioning cluster instance on an endpoint.

```
typedef struct PACK
{

    #if (defined CLD_ZLL_COMMISSION) && (defined ZLL_COMMISSION_SERVER)
        tsZCL_ClusterInstance sZllCommissionServer;
    #endif

    #if (defined CLD_ZLL_COMMISSION) && (defined ZLL_COMMISSION_CLIENT)
        tsZCL_ClusterInstance sZllCommissionClient;
    #endif

} tsZLL_CommissionEndpointClusterInstances;
```

where:

- For a Touchlink server, the following field is used:
  - `sZllCommissionServer` is a ZCL structure containing information about the ZLL Commissioning cluster server instance (refer to the *ZCL User Guide (JN-UG-3103)*).
- For a Touchlink client, the following field is used:
  - `sZllCommissionClient` is a ZCL structure containing information about the ZLL Commissioning cluster client instance (refer to the *ZCL User Guide (JN-UG-3103)*).

---

## 6.8.3 tsCLD\_ZllCommissionCustomDataStructure

This structure is used to hold the data for a Touchlink command received by a node.

```
typedef struct
{
    tsZCL_ReceiveEventAddressInterPan    sRxInterPanAddr;
    tsZCL_CallbackEvent                  sCustomCallbackEvent;
    tsCLD_ZllCommissionCallBackMessage  sCallBackMessage;
} tsCLD_ZllCommissionCustomDataStructure;
```

where:

- `RxInterPanAddr` is a ZCL structure containing the Inter-PAN addresses of the source and destination nodes of the command.
- `sCustomCallbackEvent` is the ZCL event structure for the command.

- `sCallBackMessage` is a structure containing the command ID and payload (see [Section 6.8.4](#)).

## 6.8.4 tsCLD\_ZllCommissionCallBackMessage

This structure contains the command ID and payload for a received Touchlink command.

```
typedef struct
{
    uint8    u8CommandId;
    union
    {
        tsCLD_ZllCommission_ScanReqCommandPayload
            *psScanReqPayload;
        tsCLD_ZllCommission_ScanRspCommandPayload
            *psScanRspPayload;
        tsCLD_ZllCommission_IdentifyReqCommandPayload
            *psIdentifyReqPayload;
        tsCLD_ZllCommission_DeviceInfoReqCommandPayload
            *psDeviceInfoReqPayload;
        tsCLD_ZllCommission_DeviceInfoRspCommandPayload
            *psDeviceInfoRspPayload;
        tsCLD_ZllCommission_FactoryResetReqCommandPayload
            *psFactoryResetPayload;
        tsCLD_ZllCommission_NetworkStartReqCommandPayload
            *psNwkStartReqPayload;
        tsCLD_ZllCommission_NetworkStartRspCommandPayload
            *psNwkStartRspPayload;
        tsCLD_ZllCommission_NetworkJoinRouterReqCommandPayload
            *psNwkJoinRouterReqPayload;
        tsCLD_ZllCommission_NetworkJoinRouterRspCommandPayload
            *psNwkJoinRouterRspPayload;
        tsCLD_ZllCommission_NetworkJoinEndDeviceReqCommandPayload
            *psNwkJoinEndDeviceReqPayload;
        tsCLD_ZllCommission_NetworkJoinEndDeviceRspCommandPayload
            *psNwkJoinEndDeviceRspPayload;
        tsCLD_ZllCommission_NetworkUpdateReqCommandPayload
            *psNwkUpdateReqPayload;
    } uMessage;
} tsCLD_ZllCommissionCallBackMessage;
```

where:

- `u8CommandId` is the command ID - enumerations are provided, as detailed in [Section 6.6.1](#).
- `uMessage` contains the payload of the command, where the structure used depends on the command ID (the structures are detailed in the sections below).

---

## 6.8.5 tsCLD\_ZllCommission\_ScanReqCommandPayload

This structure is used to hold the payload data for a Touchlink Scan Request command.

```
typedef struct
{
    uint32 u32TransactionId;
    uint8  u8ZigbeeInfo;
    uint8  u8ZllInfo;
} tsCLD_ZllCommission_ScanReqCommandPayload;
```

where:

- `u32TransactionId` is the 32-bit Inter-PAN Transaction Identifier of the request. This is a random number generated and inserted by the ZLL library.
- `u8ZigbeeInfo` is a bitmap of ZigBee information which indicates the ZigBee device type of the sending node and whether the radio receiver remains on when the node is idle. This information is inserted by the ZigBee stack.
- `u8ZllInfo` is a bitmap indicating the ZLL properties of the sending node, including whether the node is factory new, whether the node is able to assign addresses to other nodes and whether the node is able to initiate a link operation (supports ZLL Commissioning cluster on the client side). This information is inserted by the ZLL library.

---

## 6.8.6 tsCLD\_ZllCommission\_ScanRspCommandPayload

This structure is used to hold the payload data for a Touchlink Scan Response command.

```
typedef struct
{
    uint32 u32TransactionId;
    uint8  u8RSSICorrection;
    uint8  u8ZigbeeInfo;
    uint8  u8ZllInfo;
    uint16 u16KeyMask;
    uint32 u32ResponseId;
    uint64 u64ExtPanId;
    uint8  u8NwkUpdateId;
    uint8  u8LogicalChannel;
    uint16 u16PanId;
    uint16 u16NwkAddr;
    uint8  u8NumberSubDevices;
    uint8  u8TotalGroupIds;
    uint8  u8Endpoint;
```

```

uint16  u16ProfileId;
uint16  u16DeviceId;
uint8   u8Version;
uint8   u8GroupIdCount;
} tsCLD_ZllCommission_ScanRspCommandPayload;

```

where:

- `u32TransactionId` is the 32-bit Inter-PAN Transaction Identifier of the response, which must take the same value as the identifier in the corresponding request.
- `u8RSSICorrection` is the 8-bit RSSI correction offset for the node, in the range 0x00 to 0x20.
- `u8ZigbeeInfo` is an 8-bit field containing the following ZigBee-related information:
  - Bits 1-0: Node type (00 - Co-ordinator, 01 - Router, 10 - End Device)
  - Bit 2: Rx on when idle (1 - On, 0 - Off)
  - Bits 7-3: Reserved
- `u8ZllInfo` is an 8-bit field containing the following ZLL-related information:
  - Bit 0: Factory new (1 - Yes, 0 - No)
  - Bit 1: Address assignment capability (1 - Yes, 0 - No)
  - Bits 3-2: Reserved
  - Bit 4: Touchlink initiator (1 - Yes, 0 - No)
  - Bit 5: Touchlink priority request (1 - Yes, 0 - No)
  - Bits 7-6: Reserved
- `u16KeyMask` is a 16-bit bitmap indicating which link key is installed on the node - only one bit should be set to '1', corresponding to the key that is in use. The possible values and keys are:
  - 0x0001 (bit 0 set): Development key (defined by developer for use during application development)
  - 0x0010 (bit 4 set): Master key (obtained from the ZigBee Alliance after successful certification and agreement with the terms of the 'ZLL Security Key Licence and Confidentialty Agreement')
  - 0x8000 (bit 15 set): Certification key (defined in the ZLL Specification for use during development and during certification at test houses)
- `u32ResponseId` is a 32-bit random identifier for the response, used during network key transfer.
- `u64ExtPanId` is the 64-bit Extended PAN ID of a network to which the node already belongs, if any (a zero value indicates no network membership).
- `u8NwkUpdateId` is the current value of the Network Update Identifier on the node (see [Section 6.4.3](#)).

## Chapter 6

### ZLL Commissioning Cluster

- `u8LogicalChannel` is the number of the IEEE 802.15.4 radio channel used by a network to which the node already belongs, if any (a zero value indicates no network membership and therefore that no particular channel is used).
- `u16PanId` is the 16-bit PAN ID of a network to which the node already belongs, if any (a zero value indicates no network membership).
- `u16NwkAddr` is the 16-bit network address currently assigned to the node (the value 0xFFFF indicates that the node is 'factory new' and has no assigned network address).
- `u8NumberSubDevices` is the number of ZigBee devices on the node.
- `u8TotalGroupIds` is the total number of groups (of lights) supported on the node (across all devices).
- `u8Endpoint` is number of the endpoint (in the range 1-240) on which the ZigBee device is resident (this field is only used when there is only one ZigBee device on the node).
- `u16ProfileId` is the 16-bit identifier of the ZigBee application profile that is supported by the device (this field is only used when there is only one ZigBee device on the node).
- `u16DeviceId` is the 16-bit Device Identifier supported by the device (this field is only used when there is only one ZigBee device on the node).
- `u8Version` is an 8-bit version number for the device - the four least significant bits are from the Application Device Version field of the appropriate Simple Descriptor and the four most significant bits are zero (this field is only used when there is only one ZigBee device on the node).
- `u8GroupIdCount` is the number of groups (of lights) supported by the device (this field is only used when there is only one ZigBee device on the node).

---

#### 6.8.7 `tsCLD_ZllCommission_DeviceInfoReqCommandPayload`

This structure is used to hold the payload data for a Touchlink Device Information Request command.

```
typedef struct
{
    uint32 u32TransactionId;
    uint8  u8StartIndex;
} tsCLD_ZllCommission_DeviceInfoReqCommandPayload;
```

where:

- `u32TransactionId` is the 32-bit Inter-PAN Transaction Identifier of the request. This is a random number generated and inserted by the ZLL library.
- `u8StartIndex` specifies the index (starting from 0) of the first entry in the device table from which device information should be obtained.

---

## 6.8.8 tsCLD\_ZllCommission\_DeviceInfoRspCommandPayload

This structure is used to hold the payload data for a Touchlink Device Information Response command.

```
typedef struct
{
    uint32_t u32TransactionId;
    uint8_t u8NumberSubDevices;
    uint8_t u8StartIndex;
    uint8_t u8DeviceInfoRecordCount;
    tsCLD_ZllDeviceRecord asDeviceRecords[ZLL_MAX_DEVICE_RECORDS];
} tsCLD_ZllCommission_DeviceInfoRspCommandPayload;
```

where:

- `u32TransactionId` is the 32-bit Inter-PAN Transaction Identifier of the response, which must take the same value as the identifier in the corresponding request.
- `u8NumberSubDevices` is the number of ZigBee devices on the node (as reported in the Scan Response).
- `u8StartIndex` is the index (starting from 0) of the first entry in the device table from which device information has been obtained (this value should be as specified in the corresponding request).
- `u8DeviceInfoRecordCount` indicates the number of device information records included in the response (in the range 0 to 5).
- `asDeviceRecords[ ]` is an array, where each array element is a `tsCLD_ZllDeviceRecord` structure containing a device information record for one ZigBee device on the node (see [Section 8.2.1](#)).

---

## 6.8.9 tsCLD\_ZllCommission\_IdentifyReqCommandPayload

This structure is used to hold the payload data for a Touchlink Identify Request command.

```
typedef struct
{
    uint32_t u32TransactionId;
    uint16_t u16Duration;
} tsCLD_ZllCommission_IdentifyReqCommandPayload;
```

where:

- `u32TransactionId` is the 32-bit Inter-PAN Transaction Identifier of the request. This is a random number generated and inserted by the ZLL library.
- `u16Duration` specifies the length of time (in seconds) that the target node is to remain in identify mode. The possible values are:

- 0x0000: Exit identify mode immediately
- 0x0001–0xFFFFE: Number of seconds to remain in identify mode
- 0xFFFF: Remain in identify mode for the default time for the target node

If the target node is unable to provide accurate timings, it will attempt to remain in identify mode for as close to the requested time as possible

---

### 6.8.10 tsCLD\_ZllCommission\_FactoryResetReqCommandPayload

This structure is used to hold the payload data for a Touchlink Reset to Factory New Request command.

```
typedef struct
{
    uint32 u32TransactionId;
} tsCLD_ZllCommission_FactoryResetReqCommandPayload;
```

where `u32TransactionId` is the 32-bit Inter-PAN Transaction Identifier of the request. This is a random number generated and inserted by the ZLL library.

---

### 6.8.11 tsCLD\_ZllCommission\_NetworkStartReqCommandPayload

This structure is used to hold the payload data for a Touchlink Network Start Request command.

```
typedef struct
{
    uint32 u32TransactionId;
    uint64 u64ExtPanId;
    uint8 u8KeyIndex;
    uint8 au8NwkKey[16];
    uint8 u8LogicalChannel;
    uint16 u16PanId;
    uint16 u16NwkAddr;
    uint16 u16GroupIdBegin;
    uint16 u16GroupIdEnd;
    uint16 u16FreeNwkAddrBegin;
    uint16 u16FreeNwkAddrEnd;
    uint16 u16FreeGroupIdBegin;
    uint16 u16FreeGroupIdEnd;
    uint64 u64InitiatorIEEEAddr;
    uint16 u16InitiatorNwkAddr;
} tsCLD_ZllCommission_NetworkStartReqCommandPayload;
```

where:

- `u32TransactionId` is the 32-bit Inter-PAN Transaction Identifier of the request. This is a random number generated and inserted by the ZLL library.
- `u64ExtPanId` is the Extended PAN ID (EPID) of the new network (if set to zero, the target node will choose the EPID).
- `u8KeyIndex` is a value indicating the type of security key used to encrypt the randomly generated network key in `au8NwkKey`. The valid values are as follows (all other values are reserved for future use):
  - 0: Development key, used during development before ZLL certification
  - 4: Master key, used after successful ZLL certification
  - 15: Certification key, used during ZLL certification testing
- `au8NwkKey[16]` is the 128-bit randomly generated network key encrypted using the key specified in `u8KeyIndex`.
- `u8LogicalChannel` is the number of the IEEE 802.15.4 radio channel to be used by the network (if set to zero, the target node will choose the channel).
- `u16PanId` is the PAN ID of the new network (if set to zero, the target node will choose the PAN ID).
- `u16NwkAddr` is the 16-bit network (short) address assigned to the target node
- `u16GroupIdBegin` is the start value of the range of group identifiers that the target node can use for its own endpoints (if set to zero, no range of group identifiers has been allocated).
- `u16GroupIdEnd` is the end value of the range of group identifiers that the target node can use for its own endpoints (if set to zero, no range of group identifiers has been allocated).
- `u16FreeNwkAddrBegin` is the start address of the range of network addresses that the target node can assign to other nodes (if set to zero, no range of network addresses has been allocated).
- `u16FreeNwkAddrEnd` is the end address of the range of network addresses that the target node can assign to other nodes (if set to zero, no range of network addresses has been allocated).
- `u16FreeGroupIdBegin` is the start value of the range of free group identifiers that the target node can assign to other nodes (if set to zero, no range of free group identifiers has been allocated).
- `u16FreeGroupIdEnd` is the end value of the range of free group identifiers that the target node can assign to other nodes (if set to zero, no range of free group identifiers has been allocated).
- `u64InitiatorIEEEAddr` is the IEEE (MAC) address of the local node (network initiator)
- `u16InitiatorNwkAddr` is the network (short) address of the local node (network initiator)

---

## 6.8.12 tsCLD\_ZllCommission\_NetworkStartRspCommandPayload

This structure is used to hold the payload data for a Touchlink Network Start Response command.

```
typedef struct
{
    uint32  u32TransactionId;
    uint8   u8Status;
    uint64  u64ExtPanId;
    uint8   u8NwkUpdateId;
    uint8   u8LogicalChannel;
    uint16  u16PanId;
} tsCLD_ZllCommission_NetworkStartRspCommandPayload;
```

where:

- `u32TransactionId` is the 32-bit Inter-PAN Transaction Identifier of the response, which must take the same value as the identifier in the corresponding request.
- `u8Status` indicates the outcome of the corresponding Network Start Request: 0x00 for success, 0x01 for failure.
- `u64ExtPanId` is the Extended PAN ID (EPID) of the new network (this will be the value specified in the corresponding request or a value chosen by the local node).
- `u8NwkUpdateId` is the current value of the Network Update Identifier, which will be set to zero for a new network (see [Section 6.4.3](#)).
- `u8LogicalChannel` is the number of the IEEE 802.15.4 radio channel to be used by the network (this will be the value specified in the corresponding request or a value chosen by the local node).
- `u16PanId` is the PAN ID of the new network (this will be the value specified in the corresponding request or a value chosen by the local node).

### 6.8.13 tsCLD\_ZllCommission\_NetworkJoinRouterReqCommandPayload

This structure is used to hold the payload data for a Touchlink Network Join Router Request command.

```
typedef struct
{
    uint32  u32TransactionId;
    uint64  u64ExtPanId;
    uint8   u8KeyIndex;
    uint8   au8NwkKey[16];
    uint8   u8NwkUpdateId;
    uint8   u8LogicalChannel;
    uint16  u16PanId;
    uint16  u16NwkAddr;
    uint16  u16GroupIdBegin;
    uint16  u16GroupIdEnd;
    uint16  u16FreeNwkAddrBegin;
    uint16  u16FreeNwkAddrEnd;
    uint16  u16FreeGroupIdBegin;
    uint16  u16FreeGroupIdEnd;
} tsCLD_ZllCommission_NetworkJoinRouterReqCommandPayload;
```

where:

- `u32TransactionId` is the 32-bit Inter-PAN Transaction Identifier of the request. This is a random number generated and inserted by the ZLL library.
- `u64ExtPanId` is the Extended PAN ID (EPID) of the network.
- `u8KeyIndex` is a value indicating the type of security key used to encrypt the network key in `au8NwkKey`. The valid values are as follows (all other values are reserved for future use):
  - 0: Development key, used during development before ZLL certification
  - 4: Master key, used after successful ZLL certification
  - 15: Certification key, used during ZLL certification testing
- `au8NwkKey[16]` is the 128-bit network key encrypted using the key specified in `u8KeyIndex`.
- `u8NwkUpdateId` is the current value of the Network Update Identifier. This identifier takes a value in the range 0x00 to 0xFF and is incremented when a network update has occurred which requires the network settings on the nodes to be changed.
- `u8LogicalChannel` is the number of the IEEE 802.15.4 radio channel used by the network.
- `u16PanId` is the PAN ID of the network

- `u16NwkAddr` is the 16-bit network (short) address assigned to the target node
- `u16GroupIdBegin` is the start value of the range of group identifiers that the target node can use for its own endpoints (if set to zero, no range of group identifiers has been allocated).
- `u16GroupIdEnd` is the end value of the range of group identifiers that the target node can use for its own endpoints (if set to zero, no range of group identifiers has been allocated).
- `u16FreeNwkAddrBegin` is the start address of the range of network addresses that the target node can assign to other nodes (if set to zero, no range of network addresses has been allocated).
- `u16FreeNwkAddrEnd` is the end address of the range of network addresses that the target node can assign to other nodes (if set to zero, no range of network addresses has been allocated).
- `u16FreeGroupIdBegin` is the start value of the range of free group identifiers that the target node can assign to other nodes (if set to zero, no range of free group identifiers has been allocated).
- `u16FreeGroupIdEnd` is the end value of the range of free group identifiers that the target node can assign to other nodes (if set to zero, no range of free group identifiers has been allocated).

---

### 6.8.14 `tsCLD_ZllCommission_NetworkJoinRouterRspCommandPayload`

This structure is used to hold the payload data for a Touchlink Network Join Router Response command.

```
typedef struct
{
    uint32  u32TransactionId;
    uint8   u8Status;
} tsCLD_ZllCommission_NetworkJoinRouterRspCommandPayload;
```

where:

- `u32TransactionId` is the 32-bit Inter-PAN Transaction Identifier of the response, which must take the same value as the identifier in the corresponding request.
- `u8Status` indicates the outcome of the corresponding Network Join Router Request: 0x00 for success, 0x01 for failure.

## 6.8.15 tsCLD\_ZllCommission\_NetworkJoinEndDeviceReqCommandPayload

This structure is used to hold the payload data for a Touchlink Network Join End Device Request command.

```
typedef struct
{
    uint32  u32TransactionId;
    uint64  u64ExtPanId;
    uint8   u8KeyIndex;
    uint8   au8NwkKey[16];
    uint8   u8NwkUpdateId;
    uint8   u8LogicalChannel;
    uint16  u16PanId;
    uint16  u16NwkAddr;
    uint16  u16GroupIdBegin;
    uint16  u16GroupIdEnd;
    uint16  u16FreeNwkAddrBegin;
    uint16  u16FreeNwkAddrEnd;
    uint16  u16FreeGroupIdBegin;
    uint16  u16FreeGroupIdEnd;
} tsCLD_ZllCommission_NetworkJoinEndDeviceReqCommandPayload;
```

where:

- `u32TransactionId` is the 32-bit Inter-PAN Transaction Identifier of the request. This is a random number generated and inserted by the ZLL library.
- `u64ExtPanId` is the Extended PAN ID (EPID) of the network.
- `u8KeyIndex` is a value indicating the type of security key used to encrypt the network key in `au8NwkKey`. The valid values are as follows (all other values are reserved for future use):
  - 0: Development key, used during development before ZLL certification
  - 4: Master key, used after successful ZLL certification
  - 15: Certification key, used during ZLL certification testing
- `au8NwkKey[16]` is the 128-bit network key encrypted using the key specified in `u8KeyIndex`.
- `u8NwkUpdateId` is the current value of the Network Update Identifier. This identifier takes a value in the range 0x00 to 0xFF and is incremented when a network update has occurred which requires the network settings on the nodes to be changed.
- `u8LogicalChannel` is the number of the IEEE 802.15.4 radio channel used by the network.
- `u16PanId` is the PAN ID of the network.

- `u16NwkAddr` is the 16-bit network (short) address assigned to the target node.
- `u16GroupIdBegin` is the start value of the range of group identifiers that the target node can use for its own endpoints (if set to zero, no range of group identifiers has been allocated).
- `u16GroupIdEnd` is the end value of the range of group identifiers that the target node can use for its own endpoints (if set to zero, no range of group identifiers has been allocated).
- `u16FreeNwkAddrBegin` is the start address of the range of network addresses that the target node can assign to other nodes (if set to zero, no range of network addresses has been allocated).
- `u16FreeNwkAddrEnd` is the end address of the range of network addresses that the target node can assign to other nodes (if set to zero, no range of network addresses has been allocated).
- `u16FreeGroupIdBegin` is the start value of the range of free group identifiers that the target node can assign to other nodes (if set to zero, no range of free group identifiers has been allocated).
- `u16FreeGroupIdEnd` is the end value of the range of free group identifiers that the target node can assign to other nodes (if set to zero, no range of free group identifiers has been allocated).

---

### 6.8.16 `tsCLD_ZllCommission_NetworkJoinEndDeviceRspCommandPayload`

This structure is used to hold the payload data for a Touchlink Network Join End Device Response command.

```
typedef struct
{
    uint32 u32TransactionId;
    uint8  u8Status;
} tsCLD_ZllCommission_NetworkJoinEndDeviceRspCommandPayload;
```

where:

- `u32TransactionId` is the 32-bit Inter-PAN Transaction Identifier of the response, which must take the same value as the identifier in the corresponding request.
- `u8Status` indicates the outcome of the corresponding Network Join End Device Request: 0x00 for success, 0x01 for failure.

d

## 6.8.17 tsCLD\_ZllCommission\_NetworkUpdateReqCommandPayloa

This structure is used to hold the payload data for a Touchlink Network Update Request command.

```
typedef struct
{
    uint32  u32TransactionId;
    uint64  u64ExtPanId;
    uint8   u8NwkUpdateId;
    uint8   u8LogicalChannel;
    uint16  u16PanId;
    uint16  u16NwkAddr;
} tsCLD_ZllCommission_NetworkUpdateReqCommandPayload;
```

where:

- `u32TransactionId` is the 32-bit Inter-PAN Transaction Identifier of the request. This is a random number generated and inserted by the ZLL library.
- `u64ExtPanId` is the Extended PAN ID (EPID) of the network.
- `u8NwkUpdateId` is the current value of the Network Update Identifier (see [Section 6.4.3](#)).
- `u8LogicalChannel` is the number of the IEEE 802.15.4 radio channel used by the network.
- `u16PanId` is the PAN ID of the network.
- `u16NwkAddr` is the 16-bit network (short) address assigned to the target node.

## 6.8.18 tsCLD\_ZllUtilityCustomDataStructure

This structure is used to hold custom data for a Commissioning Utility instance of the ZLL Commissioning cluster.

```
typedef struct
{
    tsZCL_ReceiveEventAddress      sRxAddr;
    tsZCL_CallBackEvent           sCustomCallBackEvent;
    tsCLD_ZllUtilityCallBackMessage sCallBackMessage;
} tsCLD_ZllUtilityCustomDataStructure;
```

where:

- `sRxAddr` is a ZCL structure containing the destination address of the command.
- `sCustomCallBackEvent` is the ZCL event structure for the command.

- `sCallbackMessage` is a structure containing the command ID and payload (see [Section 6.8.19](#)).

---

## 6.8.19 `tsCLD_ZllUtilityCallbackMessage`

This structure contains the command ID and payload for a received Commissioning Utility command.

```
typedef struct
{
    uint8    u8CommandId;
    union
    {
        tsCLD_ZllUtility_EndpointInformationCommandPayload
            *psEndpointInfoPayload;
        tsCLD_ZllUtility_GetGroupIdReqCommandPayload
            *psGetGroupIdReqPayload;
        tsCLD_ZllUtility_GetGroupIdRspCommandPayload
            *psGetGroupIdRspPayload;
        tsCLD_ZllUtility_GetEndpointListReqCommandPayload
            *psGetEndpointlistReqPayload;
        tsCLD_ZllUtility_GetEndpointListRspCommandPayload
            *psGetEndpointListRspPayload;
    } uMessage;
} tsCLD_ZllUtilityCallbackMessage;
```

where:

- `u8CommandId` is the command ID - enumerations are provided, as detailed in [Section 6.6.2](#).
- `uMessage` contains the payload of the command, where the structure used depends on the command ID (the structures are detailed in the sections below).

---

## 6.8.20 tsCLD\_ZllUtility\_EndpointInformationCommandPayload

This structure is used to hold the payload data for a Commissioning Utility Endpoint Information command.

```
typedef struct
{
    uint64  u64IEEEAddr;
    uint16  u16NwkAddr;
    uint8   u8Endpoint;
    uint16  u16ProfileID;
    uint16  u16DeviceID;
    uint8   u8Version;
} tsCLD_ZllUtility_EndpointInformationCommandPayload;
```

where:

- `u64IEEEAddr` is the IEEE (MAC) address of the local node.
- `u16NwkAddr` is the network (short) address of the local node.
- `u8Endpoint` is the number of the local endpoint (1-240).
- `u16ProfileID` is the identifier of the ZigBee application profile supported on the local endpoint.
- `u16DeviceID` is identifier of the ZLL device on the local endpoint.
- `u8Version` specifies the version number of the ZLL device on the local endpoint.

---

## 6.9 Enumerations

---

### 6.9.1 Touchlink Event Enumerations

The event types generated by the Touchlink part of the ZLL Commissioning cluster are enumerated in the `teCLD_ZllCommission_Command` structure below:

```
typedef enum PACK
{
    E_CLD_COMMISSION_CMD_SCAN_REQ = 0x00,
    E_CLD_COMMISSION_CMD_SCAN_RSP,
    E_CLD_COMMISSION_CMD_DEVICE_INFO_REQ,
    E_CLD_COMMISSION_CMD_DEVICE_INFO_RSP,
    E_CLD_COMMISSION_CMD_IDENTIFY_REQ = 0x06,
    E_CLD_COMMISSION_CMD_FACTORY_RESET_REQ,
    E_CLD_COMMISSION_CMD_NETWORK_START_REQ = 0x10,
    E_CLD_COMMISSION_CMD_NETWORK_START_RSP,
    E_CLD_COMMISSION_CMD_NETWORK_JOIN_ROUTER_REQ,
    E_CLD_COMMISSION_CMD_NETWORK_JOIN_ROUTER_RSP,
    E_CLD_COMMISSION_CMD_NETWORK_JOIN_END_DEVICE_REQ,
    E_CLD_COMMISSION_CMD_NETWORK_JOIN_END_DEVICE_RSP,
    E_CLD_COMMISSION_CMD_NETWORK_UPDATE_REQ,
} teCLD_ZllCommission_Command;
```

---

### 6.9.2 Commissioning Utility Event Enumerations

The event types generated by the Commissioning Utility part of the ZLL Commissioning cluster are enumerated in the `teCLD_ZllUtility_Command` structure below:

```
typedef enum PACK
{
    E_CLD_UTILITY_CMD_ENDPOINT_INFO = 0x40,
    E_CLD_UTILITY_CMD_GET_GROUP_ID_REQ_RSP,
    E_CLD_UTILITY_CMD_GET_ENDPOINT_LIST_REQ_RSP,
} teCLD_ZllUtility_Command;
```

---

## 6.10 Compile-Time Options

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

The ZLL Commissioning cluster is enabled as follows:

- **Touchlink** - To enable the cluster, define `CLD_ZLL_COMMISSION`, then:
  - to enable the cluster as a server, define `ZLL_COMMISSION_SERVER`
  - to enable the cluster as a client, define `ZLL_COMMISSION_CLIENT`
- **Commissioning Utility** - To enable the cluster, define `CLD_ZLL_UTILITY`, then:
  - to enable the cluster as a server, define `ZLL_UTILITY_SERVER`
  - to enable the cluster as a client, define `ZLL_UTILITY_CLIENT`

**Chapter 6**  
**ZLL Commissioning Cluster**

# Part III: General Reference Information



## 7. ZLL Core Functions

This chapter details the core functions of the ZigBee Light Link API. These comprise the following initialisation function, timing update function and device-specific endpoint registration functions:

Function	Page
<a href="#">eZLL_Initialise</a>	128
<a href="#">eZLL_Update100mS</a>	129
<a href="#">eZLL_RegisterOnOffLightEndPoint</a>	130
<a href="#">eZLL_RegisterOnOffPlugEndPoint</a>	132
<a href="#">eZLL_RegisterDimmableLightEndPoint</a>	134
<a href="#">eZLL_RegisterDimmablePlugEndPoint</a>	136
<a href="#">eZLL_RegisterColourLightEndPoint</a>	138
<a href="#">eZLL_RegisterExtendedColourLightEndPoint</a>	140
<a href="#">eZLL_RegisterColourTemperatureLightEndPoint</a>	142
<a href="#">eZLL_RegisterColourRemoteEndPoint</a>	144
<a href="#">eZLL_RegisterColourSceneRemoteEndPoint</a>	146
<a href="#">eZLL_RegisterNonColourRemoteEndPoint</a>	148
<a href="#">eZLL_RegisterNonColourSceneRemoteEndPoint</a>	150
<a href="#">eZLL_RegisterControlBridgeEndPoint</a>	152
<a href="#">eZLL_RegisterOnOffSensorEndPoint</a>	154



**Note 1:** For guidance on using these functions in your application code, refer to [Chapter 4](#).

**Note 2:** The return codes for these functions are described in the *ZCL User Guide (JN-UG-3103)*.

**Note 3:** ZLL initialisation must also be performed through definitions in the header file **zcl\_options.h** - see [Section 3.5.1](#). In addition, JenOS resources for ZLL must also be pre-configured using the JenOS Configuration Editor - refer to the *JenOS User Guide (JN-UG-3075)*.

---

## eZLL\_Initialise

---

```
teZCL_Status eZLL_Initialise(  
    tfpZCL_ZCLCallbackFunction cbCallback,  
    PDUM_thAPdu hAPdu);
```

### Description

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

As part of this function call, you must specify a user-defined callback function that will be invoked when a ZigBee PRO stack event occurs that is not associated with an endpoint (the callback function for events associated with an endpoint is specified when the endpoint is registered using one of the registration functions). This callback function is defined according to the typedef:

```
typedef void (* tfpZCL_ZCLCallbackFunction)  
    (tsZCL_CallbackEvent *pCallbackEvent);
```

You must also provide a pointer to a local pool of Application Protocol Data Units (APDUs) that will be used by the ZCL to hold messages to be sent and received.

### Parameters

<i>cbCallback</i>	Pointer to a callback function to handle stack events that are not associated with a registered endpoint
<i>hAPdu</i>	Pointer to a pool of APDUs for holding messages to be sent and received

### Returns

E\_ZCL\_SUCCESS  
E\_ZCL\_ERR\_HEAP\_FAIL  
E\_ZCL\_ERR\_PARAMETER\_NULL

---

## eZLL\_Update100mS

---

```
teZCL_Status eZLL_Update100mS(void);
```

### Description

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

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

```
void vIdEffectTick(void)
```

### Parameters

None

### Returns

E\_ZCL\_SUCCESS

---

## eZLL\_RegisterOnOffLightEndPoint

---

```
teZCL_Status eZLL_RegisterOnOffLightEndPoint(  
    uint8 u8EndPointIdentifier,  
    tfpZCL_ZCLCallbackFunction cbCallback,  
    tsZLL_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 the **eZLL\_Initialise()** function and before starting the ZigBee PRO stack.

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

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 `tsZLL_OnOffLightDevice` structure (see [Section 8.1.1](#)) 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 JN51xx module.

### Parameters

<i>u8EndPointIdentifier</i>	Identifier of endpoint to be registered - this is an endpoint number in the range 1 to 240
<i>cbCallback</i>	Pointer to a callback function to handle events associated with the registered endpoint
<i>psDeviceInfo</i>	Pointer to structure to be used to hold On/Off Light device variables

## Returns

E\_ZCL\_SUCCESS  
E\_ZCL\_FAIL  
E\_ZCL\_ERR\_PARAMETER\_NULL  
E\_ZCL\_ERR\_PARAMETER\_RANGE  
E\_ZCL\_ERR\_EP\_RANGE  
E\_ZCL\_ERR\_CLUSTER\_0  
E\_ZCL\_ERR\_CALLBACK\_NULL

---

## eZLL\_RegisterOnOffPlugEndPoint

---

```
teZCL_Status eZLL_RegisterOnOffPlugEndPoint(  
    uint8 u8EndPointIdentifier,  
    tfpZCL_ZCLCallbackFunction cbCallback,  
    tsZLL_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 the **eZLL\_Initialise()** function and before starting the ZigBee PRO stack.

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

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 `tsZLL_OnOffPlugDevice` structure (see [Section 8.1.2](#)) 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 JN51xx module.

### Parameters

<i>u8EndPointIdentifier</i>	Identifier of endpoint to be registered - this is an endpoint number in the range 1 to 240
<i>cbCallback</i>	Pointer to a callback function to handle events associated with the registered endpoint
<i>psDeviceInfo</i>	Pointer to structure to be used to hold On/Off Plug-in Unit device variables

## Returns

E\_ZCL\_SUCCESS  
E\_ZCL\_FAIL  
E\_ZCL\_ERR\_PARAMETER\_NULL  
E\_ZCL\_ERR\_PARAMETER\_RANGE  
E\_ZCL\_ERR\_EP\_RANGE  
E\_ZCL\_ERR\_CLUSTER\_0  
E\_ZCL\_ERR\_CALLBACK\_NULL

---

## eZLL\_RegisterDimmableLightEndPoint

---

```
teZCL_Status eZLL_RegisterDimmableLightEndPoint(  
    uint8 u8EndPointIdentifier,  
    tfpZCL_ZCLCallbackFunction cbCallBack,  
    tsZLL_DimmableLightDevice *psDeviceInfo);
```

### Description

This function is used to register an endpoint which will support a Dimmable Light device. The function must be called after the **eZLL\_Initialise()** function and before starting the ZigBee PRO stack.

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

As part of this function call, you must specify a user-defined callback function that will be invoked when an event occurs that is associated with the endpoint (events are detailed in the *ZCL User Guide (JN-UG-3103)*). This callback function is defined according to the typedef:

```
typedef void (* tfpZCL_ZCLCallbackFunction)  
    (tsZCL_CallbackEvent *pCallbackEvent);
```

You must also provide a pointer to a `tsZLL_DimmableLightDevice` structure (see [Section 8.1.3](#)) 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 JN51xx module.

### Parameters

<i>u8EndPointIdentifier</i>	Identifier of endpoint to be registered - this is an endpoint number in the range 1 to 240
<i>cbCallBack</i>	Pointer to a callback function to handle events associated with the registered endpoint
<i>psDeviceInfo</i>	Pointer to structure to be used to hold Dimmable Light device variables

## Returns

E\_ZCL\_SUCCESS  
E\_ZCL\_FAIL  
E\_ZCL\_ERR\_PARAMETER\_NULL  
E\_ZCL\_ERR\_PARAMETER\_RANGE  
E\_ZCL\_ERR\_EP\_RANGE  
E\_ZCL\_ERR\_CLUSTER\_0  
E\_ZCL\_ERR\_CALLBACK\_NULL

---

## eZLL\_RegisterDimmablePlugEndPoint

---

```
teZCL_Status eZLL_RegisterDimmablePlugEndPoint(
    uint8 u8EndPointIdentifier,
    tfpZCL_ZCLCallbackFunction cbCallBack,
    tsZLL_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 the **eZLL\_Initialise()** function and before starting the ZigBee PRO stack.

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

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 `tsZLL_DimmablePlugDevice` structure (see [Section 8.1.4](#)) 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 JN51xx module.

### Parameters

<i>u8EndPointIdentifier</i>	Identifier of endpoint to be registered - this is an endpoint number in the range 1 to 240
<i>cbCallBack</i>	Pointer to a callback function to handle events associated with the registered endpoint
<i>psDeviceInfo</i>	Pointer to structure to be used to hold Dimmable Plug-in Unit device variables

## Returns

E\_ZCL\_SUCCESS  
E\_ZCL\_FAIL  
E\_ZCL\_ERR\_PARAMETER\_NULL  
E\_ZCL\_ERR\_PARAMETER\_RANGE  
E\_ZCL\_ERR\_EP\_RANGE  
E\_ZCL\_ERR\_CLUSTER\_0  
E\_ZCL\_ERR\_CALLBACK\_NULL

---

## eZLL\_RegisterColourLightEndPoint

---

```
teZCL_Status eZLL_RegisterColourLightEndPoint(  
    uint8 u8EndPointIdentifier,  
    tfpZCL_ZCLCallbackFunction cbCallback,  
    tsZLL_ColourLightDevice *psDeviceInfo);
```

### Description

This function is used to register an endpoint which will support a Colour Light device. The function must be called after the **eZLL\_Initialise()** function and before starting the ZigBee PRO stack.

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

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 `tsZLL_ColourLightDevice` structure (see [Section 8.1.5](#)) which will be used to store all variables relating to the 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 Colour Light device is housed in the same hardware, sharing the same JN51xx module.

### Parameters

<i>u8EndPointIdentifier</i>	Identifier of endpoint to be registered - this is an endpoint number in the range 1 to 240
<i>cbCallback</i>	Pointer to a callback function to handle events associated with the registered endpoint
<i>psDeviceInfo</i>	Pointer to structure to be used to hold Colour Light device variables

## Returns

E\_ZCL\_SUCCESS  
E\_ZCL\_FAIL  
E\_ZCL\_ERR\_PARAMETER\_NULL  
E\_ZCL\_ERR\_PARAMETER\_RANGE  
E\_ZCL\_ERR\_EP\_RANGE  
E\_ZCL\_ERR\_CLUSTER\_0  
E\_ZCL\_ERR\_CALLBACK\_NULL

---

## eZLL\_RegisterExtendedColourLightEndPoint

---

```
teZCL_Status eZLL_RegisterExtendedColourLightEndPoint(  
    uint8 u8EndPointIdentifier,  
    tfpZCL_ZCLCallbackFunction cbCallBack,  
    tsZLL_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 the **eZLL\_Initialise()** function and before starting the ZigBee PRO stack.

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

As part of this function call, you must specify a user-defined callback function that will be invoked when an event occurs that is associated with the endpoint (events are detailed in the *ZCL User Guide (JN-UG-3103)*). This callback function is defined according to the typedef:

```
typedef void (* tfpZCL_ZCLCallbackFunction)  
    (tsZCL_CallbackEvent *pCallBackEvent);
```

You must also provide a pointer to a `tsZLL_ExtendedColourLightDevice` structure (see [Section 8.1.6](#)) 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 JN51xx module.

### Parameters

<i>u8EndPointIdentifier</i>	Identifier of endpoint to be registered - this is an endpoint number in the range 1 to 240
<i>cbCallBack</i>	Pointer to a callback function to handle events associated with the registered endpoint
<i>psDeviceInfo</i>	Pointer to structure to be used to hold Extended Colour Light device variables

## Returns

E\_ZCL\_SUCCESS  
E\_ZCL\_FAIL  
E\_ZCL\_ERR\_PARAMETER\_NULL  
E\_ZCL\_ERR\_PARAMETER\_RANGE  
E\_ZCL\_ERR\_EP\_RANGE  
E\_ZCL\_ERR\_CLUSTER\_0  
E\_ZCL\_ERR\_CALLBACK\_NULL

---

## eZLL\_RegisterColourTemperatureLightEndPoint

---

```
teZCL_Status  
eZLL_RegisterColourTemperatureLightEndPoint(  
    uint8 u8EndPointIdentifier,  
    tfpZCL_ZCLCallbackFunction cbCallBack,  
    tsZLL_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 the **eZLL\_Initialise()** function and before starting the ZigBee PRO stack.

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

As part of this function call, you must specify a user-defined callback function that will be invoked when an event occurs that is associated with the endpoint (events are detailed in the *ZCL User Guide (JN-UG-3103)*). This callback function is defined according to the typedef:

```
typedef void (* tfpZCL_ZCLCallbackFunction)  
    (tsZCL_CallbackEvent *pCallbackEvent);
```

You must also provide a pointer to a `tsZLL_ColourTemperatureLightDevice` structure (see [Section 8.1.7](#)) 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 JN51xx module.

### Parameters

<i>u8EndPointIdentifier</i>	Identifier of endpoint to be registered - this is an endpoint number in the range 1 to 240
<i>cbCallBack</i>	Pointer to a callback function to handle events associated with the registered endpoint
<i>psDeviceInfo</i>	Pointer to structure to be used to hold Colour Temperature Light device variables

## Returns

E\_ZCL\_SUCCESS  
E\_ZCL\_FAIL  
E\_ZCL\_ERR\_PARAMETER\_NULL  
E\_ZCL\_ERR\_PARAMETER\_RANGE  
E\_ZCL\_ERR\_EP\_RANGE  
E\_ZCL\_ERR\_CLUSTER\_0  
E\_ZCL\_ERR\_CALLBACK\_NULL

---

## eZLL\_RegisterColourRemoteEndPoint

---

```
teZCL_Status eZLL_RegisterColourRemoteEndPoint(
    uint8 u8EndPointIdentifier,
    tfpZCL_ZCLCallbackFunction cbCallBack,
    tsZLL_ColourRemoteDevice *psDeviceInfo);
```

### Description

This function is used to register an endpoint which will support a Colour Controller device. The function must be called after the **eZLL\_Initialise()** function and before starting the ZigBee PRO stack.

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

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 `tsZLL_ColourRemoteDevice` structure (see [Section 8.1.8](#)) 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 JN51xx module.

### Parameters

<i>u8EndPointIdentifier</i>	Identifier of endpoint to be registered - this is an endpoint number in the range 1 to 240
<i>cbCallBack</i>	Pointer to a callback function to handle events associated with the registered endpoint
<i>psDeviceInfo</i>	Pointer to structure to be used to hold Colour Controller device variables

## Returns

E\_ZCL\_SUCCESS  
E\_ZCL\_FAIL  
E\_ZCL\_ERR\_PARAMETER\_NULL  
E\_ZCL\_ERR\_PARAMETER\_RANGE  
E\_ZCL\_ERR\_EP\_RANGE  
E\_ZCL\_ERR\_CLUSTER\_0  
E\_ZCL\_ERR\_CALLBACK\_NULL

---

## eZLL\_RegisterColourSceneRemoteEndPoint

---

```
teZCL_Status eZLL_RegisterColourSceneRemoteEndPoint(  
    uint8 u8EndPointIdentifier,  
    tfpZCL_ZCLCallbackFunction cbCallBack,  
    tsZLL_ColourSceneRemoteDevice *psDeviceInfo);
```

### Description

This function is used to register an endpoint which will support a Colour Scene Controller device. The function must be called after the **eZLL\_Initialise()** function and before starting the ZigBee PRO stack.

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

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 `tsZLL_ColourSceneRemoteDevice` structure (see [Section 8.1.9](#)) 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 JN51xx module.

### Parameters

<i>u8EndPointIdentifier</i>	Identifier of endpoint to be registered - this is an endpoint number in the range 1 to 240
<i>cbCallBack</i>	Pointer to a callback function to handle events associated with the registered endpoint
<i>psDeviceInfo</i>	Pointer to structure to be used to hold Colour Scene Controller device variables

## Returns

E\_ZCL\_SUCCESS  
E\_ZCL\_FAIL  
E\_ZCL\_ERR\_PARAMETER\_NULL  
E\_ZCL\_ERR\_PARAMETER\_RANGE  
E\_ZCL\_ERR\_EP\_RANGE  
E\_ZCL\_ERR\_CLUSTER\_0  
E\_ZCL\_ERR\_CALLBACK\_NULL

---

## eZLL\_RegisterNonColourRemoteEndPoint

---

```
teZCL_Status eZLL_RegisterNonColourRemoteEndPoint(  
    uint8 u8EndPointIdentifier,  
    tfpZCL_ZCLCallbackFunction cbCallBack,  
    tsZLL_NonColourRemoteDevice *psDeviceInfo);
```

### Description

This function is used to register an endpoint which will support a Non-Colour Controller device. The function must be called after the **eZLL\_Initialise()** function and before starting the ZigBee PRO stack.

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

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 `tsZLL_NonColourRemoteDevice` structure (see [Section 8.1.10](#)) 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 JN51xx module.

### Parameters

<i>u8EndPointIdentifier</i>	Identifier of endpoint to be registered - this is an endpoint number in the range 1 to 240
<i>cbCallBack</i>	Pointer to a callback function to handle events associated with the registered endpoint
<i>psDeviceInfo</i>	Pointer to structure to be used to hold Non-Colour Controller device variables

## Returns

E\_ZCL\_SUCCESS  
E\_ZCL\_FAIL  
E\_ZCL\_ERR\_PARAMETER\_NULL  
E\_ZCL\_ERR\_PARAMETER\_RANGE  
E\_ZCL\_ERR\_EP\_RANGE  
E\_ZCL\_ERR\_CLUSTER\_0  
E\_ZCL\_ERR\_CALLBACK\_NULL

---

## eZLL\_RegisterNonColourSceneRemoteEndPoint

---

```
teZCL_Status  
eZLL_RegisterNonColourSceneRemoteEndPoint(  
    uint8 u8EndPointIdentifier,  
    tfpZCL_ZCLCallbackFunction cbCallBack,  
    tsZLL_NonColourSceneRemoteDevice  
        *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 the **eZLL\_Initialise()** function and before starting the ZigBee PRO stack.

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

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 `tsZLL_NonColourSceneRemoteDevice` structure (see [Section 8.1.11](#)) 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 JN51xx module.

### Parameters

<i>u8EndPointIdentifier</i>	Identifier of endpoint to be registered - this is an endpoint number in the range 1 to 240
<i>cbCallBack</i>	Pointer to a callback function to handle events associated with the registered endpoint
<i>psDeviceInfo</i>	Pointer to structure to be used to hold Non-Colour Scene Controller device variables

## Returns

E\_ZCL\_SUCCESS  
E\_ZCL\_FAIL  
E\_ZCL\_ERR\_PARAMETER\_NULL  
E\_ZCL\_ERR\_PARAMETER\_RANGE  
E\_ZCL\_ERR\_EP\_RANGE  
E\_ZCL\_ERR\_CLUSTER\_0  
E\_ZCL\_ERR\_CALLBACK\_NULL

---

## eZLL\_RegisterControlBridgeEndPoint

---

```
teZCL_Status eZLL_RegisterControlBridgeEndPoint(  
    uint8 u8EndPointIdentifier,  
    tfpZCL_ZCLCallbackFunction cbCallBack,  
    tsZLL_ControlBridgeDevice *psDeviceInfo);
```

### Description

This function is used to register an endpoint which will support a Control Bridge device. The function must be called after the **eZLL\_Initialise()** function and before starting the ZigBee PRO stack.

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

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 `tsZLL_ControlBridgeDevice` structure (see [Section 8.1.12](#)) 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 JN51xx module.

### Parameters

<i>u8EndPointIdentifier</i>	Identifier of endpoint to be registered - this is an endpoint number in the range 1 to 240
<i>cbCallBack</i>	Pointer to a callback function to handle events associated with the registered endpoint
<i>psDeviceInfo</i>	Pointer to structure to be used to hold Control Bridge device variables

## Returns

E\_ZCL\_SUCCESS  
E\_ZCL\_FAIL  
E\_ZCL\_ERR\_PARAMETER\_NULL  
E\_ZCL\_ERR\_PARAMETER\_RANGE  
E\_ZCL\_ERR\_EP\_RANGE  
E\_ZCL\_ERR\_CLUSTER\_0  
E\_ZCL\_ERR\_CALLBACK\_NULL

---

## eZLL\_RegisterOnOffSensorEndPoint

---

```
teZCL_Status eZLL_RegisterOnOffSensorEndPoint(  
    uint8 u8EndPointIdentifier,  
    tfpZCL_ZCLCallbackFunction cbCallBack,  
    tsZLL_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 the **eZLL\_Initialise()** function and before starting the ZigBee PRO stack.

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

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 `tsZLL_OnOffSensorDevice` structure (see [Section 8.1.13](#)) 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 JN51xx module.

### Parameters

<i>u8EndPointIdentifier</i>	Identifier of endpoint to be registered - this is an endpoint number in the range 1 to 240
<i>cbCallBack</i>	Pointer to a callback function to handle events associated with the registered endpoint
<i>psDeviceInfo</i>	Pointer to structure to be used to hold On/Off Sensor device variables

## Returns

E\_ZCL\_SUCCESS  
E\_ZCL\_FAIL  
E\_ZCL\_ERR\_PARAMETER\_NULL  
E\_ZCL\_ERR\_PARAMETER\_RANGE  
E\_ZCL\_ERR\_EP\_RANGE  
E\_ZCL\_ERR\_CLUSTER\_0  
E\_ZCL\_ERR\_CALLBACK\_NULL

**Chapter 7**  
**ZLL Core Functions**

---

## 8. ZLL Structures

This chapter presents general ZLL structures (that is, not associated with any particular cluster). ZLL Commissioning cluster structures are detailed in [Chapter 6](#).

---

### 8.1 Device Structures

The shared device structures for the ZLL devices supported by the ZLL API are presented below. Within each shared device structure, there is a section for each cluster supported by the device, where each of these sections has one or more of the following elements:

- Pointer to the cluster
- Data structure(s) for the cluster

The section for each optional cluster is enabled by a corresponding enumeration defined in the **zcl\_options.h** file (e.g. CLD\_SCENES for the Scenes cluster). Another enumeration is also used which determines whether the cluster will act as a server or client (e.g. SCENES\_SERVER for a Scenes cluster server). Refer to [Section 3.5.1](#).

---

#### 8.1.1 tsZLL\_OnOffLightDevice

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

```
typedef struct
{
    tsZCL_EndPointDefinition sEndPoint;

    /* Cluster instances */
    tsZLL_OnOffLightDeviceClusterInstances sClusterInstance;

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

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

#ifdef (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
} tsZLL_OnOffLightDevice;
```

---

## 8.1.2 tsZLL\_OnOffPlugDevice

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

```
typedef struct
{
    tsZCL_EndPointDefinition sEndPoint;

    /* Cluster instances */
    tsZLL_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

} tsZLL_OnOffPlugDevice;
```

---

### 8.1.3 tsZLL\_DimmableLightDevice

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

```
typedef struct
{
    tsZCL_EndPointDefinition sEndPoint;

    /* Cluster instances */
    tsZLL_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

} tsZLL_DimmableLightDevice;
```

---

## 8.1.4 tsZLL\_DimmablePlugDevice

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

```
typedef struct
{
    tsZCL_EndPointDefinition sEndPoint;

    /* Cluster instances */
    tsZLL_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
```

```
#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

} tsZLL_DimmablePlugDevice;
```

---

### 8.1.5 tsZLL\_ColourLightDevice

The following tsZLL\_ColourLightDevice structure is the shared structure for a Colour Light device:

```
typedef struct
{
    tsZCL_EndPointDefinition sEndPoint;

    /* Cluster instances */
    tsZLL_ColourLightDeviceClusterInstances 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

} tsZLL_ColourLightDevice;

```

---

### 8.1.6 tsZLL\_ExtendedColourLightDevice

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

```

typedef struct
{
    tsZCL_EndPointDefinition sEndPoint;

    /* Cluster instances */

```

## Chapter 8

### ZLL Structures

```
    tsZLL_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  
  
} tsZLL_ExtendedColourLightDevice;
```

---

## 8.1.7 tsZLL\_ColourTemperatureLightDevice

The following tsZLL\_ColourTemperatureLightDevice structure is the shared structure for a Colour Temperature Light device:

```
typedef struct
{
    tsZCL_EndPointDefinition sEndPoint;

    /* Cluster instances */
    tsZLL_ColourTemperatureLightDeviceClusterInstances
    sClusterInstance;

#ifdef CLD_BASIC && BASIC_SERVER
    /* Basic Cluster - Server */
    tsCLD_Basic sBasicServerCluster;
#endif

#ifdef CLD_IDENTIFY && IDENTIFY_SERVER
    /* Identify Cluster - Server */
    tsCLD_Identify sIdentifyServerCluster;
    tsCLD_IdentifyCustomDataStructure
    sIdentifyServerCustomDataStructure;
#endif

#ifdef CLD_ONOFF && ONOFF_SERVER
    /* On/Off Cluster - Server */
    tsCLD_OnOff sOnOffServerCluster;
    tsCLD_OnOffCustomDataStructure
    sOnOffServerCustomDataStructure;
#endif

#ifdef CLD_GROUPS && GROUPS_SERVER
    /* Groups Cluster - Server */
    tsCLD_Groups sGroupsServerCluster;
    tsCLD_GroupsCustomDataStructure
    sGroupsServerCustomDataStructure;
#endif

#ifdef CLD_SCENES && 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

} tsZLL_ColourTemperatureLightDevice;

```

---

### 8.1.8 tsZLL\_ColourRemoteDevice

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

```

typedef struct
{
    tsZCL_EndPointDefinition sEndPoint;

    /* Cluster instances */
    tsZLL_ColourRemoteDeviceClusterInstances sClusterInstance;

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

```

## Chapter 8 ZLL Structures

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

    #if (defined CLD_ZLL_UTILITY) && (defined ZLL_UTILITY_SERVER)
        /* Utility Cluster - Server */
        tsCLD_ZllUtility sZllUtilityServerCluster;
        tsCLD_ZllUtilityCustomDataStructure
    sZllUtilityServerCustomDataStructure;
    #endif

    #if (defined CLD_ZLL_UTILITY) && (defined ZLL_UTILITY_CLIENT)
        /* Utility Cluster - Client */
        tsCLD_ZllUtility sZllUtilityClientCluster;
        tsCLD_ZllUtilityCustomDataStructure
    sZllUtilityClientCustomDataStructure;
    #endif

} tsZLL_ColourRemoteDevice;
```

---

## 8.1.9 tsZLL\_ColourSceneRemoteDevice

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

```
typedef struct
{
    tsZCL_EndPointDefinition sEndPoint;

    /* Cluster instances */
    tsZLL_ColourSceneRemoteDeviceClusterInstances sClusterInstance;

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

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

#if (defined CLD_ZLL_UTILITY) && (defined ZLL_UTILITY_SERVER)
    tsCLD_ZllUtility sZllUtilityServerCluster;
    tsCLD_ZllUtilityCustomDataStructure
sZllUtilityServerCustomDataStructure;
#endif

#if (defined CLD_ZLL_UTILITY) && (defined ZLL_UTILITY_CLIENT)
    tsCLD_ZllUtility sZllUtilityClientCluster;
    tsCLD_ZllUtilityCustomDataStructure
sZllUtilityClientCustomDataStructure;
#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 */
```

## Chapter 8 ZLL Structures

```
        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

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

---

### 8.1.10 tsZLL\_NonColourRemoteDevice

The following `tsZLL_NonColourRemoteDevice` structure is the shared structure for a Non-Colour Controller device:

```
typedef struct
{
    tsZCL_EndPointDefinition sEndPoint;

    /* Cluster instances */
    tsZLL_NonColourRemoteDeviceClusterInstances sClusterInstance;

    #if (defined CLD_BASIC) && (defined BASIC_SERVER)
```

```

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

#if (defined CLD_ZLL_UTILITY) && (defined ZLL_UTILITY_SERVER)
    /* Utility Cluster - Server */
    tsCLD_ZllUtility sZllUtilityServerCluster;
    tsCLD_ZllUtilityCustomDataStructure
sZllUtilityServerCustomDataStructure;
#endif

#if (defined CLD_ZLL_UTILITY) && (defined ZLL_UTILITY_CLIENT)
    /* Utility Cluster - Client */
    tsCLD_ZllUtility sZllUtilityClientCluster;
    tsCLD_ZllUtilityCustomDataStructure
sZllUtilityClientCustomDataStructure;
#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

#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

```

```
} tsZLL_NonColourRemoteDevice;
```

---

### 8.1.11 tsZLL\_NonColourSceneRemoteDevice

The following `tsZLL_NonColourSceneRemoteDevice` structure is the shared structure for a Non-Colour Scene Controller device:

```
typedef struct
{
    tsZCL_EndPointDefinition sEndPoint;

    /* Cluster instances */
    tsZLL_NonColourSceneRemoteDeviceClusterInstances
sClusterInstance;

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

#if (defined CLD_ZLL_UTILITY) && (defined ZLL_UTILITY_SERVER)
    /* Utility Cluster - Server */
    tsCLD_ZllUtility sZllUtilityServerCluster;
    tsCLD_ZllUtilityCustomDataStructure
sZllUtilityServerCustomDataStructure;
#endif

#if (defined CLD_ZLL_UTILITY) && (defined ZLL_UTILITY_CLIENT)
    /* Utility Cluster - Client */
    tsCLD_ZllUtility sZllUtilityClientCluster;
    tsCLD_ZllUtilityCustomDataStructure
sZllUtilityClientCustomDataStructure;
#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

    #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

} tsZLL_NonColourSceneRemoteDevice;

```

---

## 8.1.12 tsZLL\_ControlBridgeDevice

The following `tsZLL_ControlBridgeDevice` structure is the shared structure for a Control Bridge device:

```

typedef struct
{
    tsZCL_EndPointDefinition sEndPoint;

    /* Cluster instances */
    tsZLL_ControlBridgeDeviceClusterInstances sClusterInstance;

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

    #if (defined CLD_ZLL_UTILITY) && (defined ZLL_UTILITY_SERVER)

```

## Chapter 8 ZLL Structures

```
        /* Utility Cluster - Server */
        tsCLD_ZllUtility                szllUtilityServerCluster;
        tsCLD_ZllUtilityCustomDataStructure
sZllUtilityServerCustomDataStructure;
    #endif

    #if (defined CLD_ZLL_UTILILITY) && (defined ZLL_UTILILITY_CLIENT)
        /* Utility Cluster - Client */
        tsCLD_ZllUtility                szllUtilityClientCluster;
        tsCLD_ZllUtilityCustomDataStructure
sZllUtilityClientCustomDataStructure;
    #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

    #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

    /* Mandatory client clusters */
    #if (defined CLD_DOOR_LOCK) && (defined DOOR_LOCK_CLIENT)
        /* Door Lock Cluster - Client */
        tsCLD_DoorLock sDoorLockClientCluster;
    #endif

} tsZLL_ControlBridgeDevice;

```

---

### 8.1.13 tsZLL\_OnOffSensorDevice

The following tsZLL\_OnOffSensorDevice structure is the shared structure for a On/Off Sensor device:

```

typedef struct
{
    tsZCL_EndPointDefinition sEndPoint;

    /* Cluster instances */
    tsZLL_OnOffSensorDeviceClusterInstances sClusterInstance;

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

    #if (defined CLD_ZLL_UTILITY) && (defined ZLL_UTILITY_SERVER)
        /* Utility Cluster - Server */
        tsCLD_ZllUtility sZllUtilityServerCluster;
        tsCLD_ZllUtilityCustomDataStructure
sZllUtilityServerCustomDataStructure;
    #endif
}

```

## Chapter 8 ZLL Structures

```
#if (defined CLD_ZLL_UTILITY) && (defined ZLL_UTILITY_CLIENT)
    /* Utility Cluster - Client */
    tsCLD_ZllUtility sZllUtilityClientCluster;
    tsCLD_ZllUtilityCustomDataStructure
sZllUtilityClientCustomDataStructure;
#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

#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

} tsZLL_OnOffSensorDevice;
```

---

## 8.2 Other Structures

---

### 8.2.1 tsCLD\_ZllDeviceRecord

The following `tsCLD_ZllDeviceRecord` structure is used to store a device information record about a ZigBee device on a node:

```
typedef struct
{
    uint64    u64IEEEAddr;
    uint16    u16ProfileId;
    uint16    u16DeviceId;
    uint8     u8Endpoint;
    uint8     u8Version;
    uint8     u8NumberGroupIds;
    uint8     u8Sort;
} tsCLD_ZllDeviceRecord;
```

where:

- `u64IEEEAddr` is the 64-bit IEEE (MAC) address of the node.
- `u16ProfileId` is the 16-bit identifier of the ZigBee application profile that is supported by the device.
- `u16DeviceId` is the 16-bit Device Identifier supported by the device.
- `u8Endpoint` is number of the endpoint (in the range 1-240) on which the device is located
- `u8Version` is an 8-bit version number for the device - the four least significant bits are from the Application Device Version field of the appropriate Simple Descriptor and the four most significant bits are zero.
- `u8NumberGroupIds` is the number of groups (of lights) supported by the device.
- `u8Sort` indicates the position of the device in a sorted list of devices, e.g. for a remote control unit (a zero value indicates that there is no sorted list).

## Revision History

Version	Date	Comments
1.0	24-Jan-2013	First release
1.1	14-Aug-2013	Various updates for new ZigBee Light Link release
1.2	24-Feb-2015	Updated for change of toolchain to 'BeyondStudio for NXP' and for optimised ZCL supplied in new combined HA/ZLL SDK. Part numbers for related resources changed: <ul style="list-style-type: none"><li>• ZigBee PRO Stack User Guide: JN-UG-3048 to JN-UG-3101</li><li>• ZCL User Guide: JN-UG-3077 to JN-UG-3103</li><li>• JN516x Toolchain installer: JN-SW-4041 to JN-SW-4141</li><li>• JN516x ZLL SDK installer: JN-SW-4062 to JN-SW-4168</li></ul>
1.3	3-Aug-2016	Web addresses for NXP Wireless Connectivity pages updated

## Important Notice

**Limited warranty and liability** - Information in this document is believed to be accurate and reliable. However, NXP Semiconductors does not give any representations or warranties, expressed or implied, as to the accuracy or completeness of such information and shall have no liability for the consequences of use of such information. NXP Semiconductors takes no responsibility for the content in this document if provided by an information source outside of NXP Semiconductors.

In no event shall NXP Semiconductors be liable for any indirect, incidental, punitive, special or consequential damages (including - without limitation - lost profits, lost savings, business interruption, costs related to the removal or replacement of any products or rework charges) whether or not such damages are based on tort (including negligence), warranty, breach of contract or any other legal theory.

Notwithstanding any damages that customer might incur for any reason whatsoever, NXP Semiconductors' aggregate and cumulative liability towards customer for the products described herein shall be limited in accordance with the *Terms and conditions of commercial sale* of NXP Semiconductors.

**Right to make changes** - NXP Semiconductors reserves the right to make changes to information published in this document, including without limitation specifications and product descriptions, at any time and without notice. This document supersedes and replaces all information supplied prior to the publication hereof.

**Suitability for use** - NXP Semiconductors products are not designed, authorized or warranted to be suitable for use in life support, life-critical or safety-critical systems or equipment, nor in applications where failure or malfunction of an NXP Semiconductors product can reasonably be expected to result in personal injury, death or severe property or environmental damage. NXP Semiconductors and its suppliers accept no liability for inclusion and/or use of NXP Semiconductors products in such equipment or applications and therefore such inclusion and/or use is at the customer's own risk.

**Applications** - Applications that are described herein for any of these products are for illustrative purposes only. NXP Semiconductors makes no representation or warranty that such applications will be suitable for the specified use without further testing or modification.

Customers are responsible for the design and operation of their applications and products using NXP Semiconductors products, and NXP Semiconductors accepts no liability for any assistance with applications or customer product design. It is customer's sole responsibility to determine whether the NXP Semiconductors product is suitable and fit for the customer's applications and products planned, as well as for the planned application and use of customer's third party customer(s). Customers should provide appropriate design and operating safeguards to minimize the risks associated with their applications and products.

NXP Semiconductors does not accept any liability related to any default, damage, costs or problem which is based on any weakness or default in the customer's applications or products, or the application or use by customer's third party customer(s). Customer is responsible for doing all necessary testing for the customer's applications and products using NXP Semiconductors products in order to avoid a default of the applications and the products or of the application or use by customer's third party customer(s). NXP does not accept any liability in this respect.

**Export control** - This document as well as the item(s) described herein may be subject to export control regulations. Export might require a prior authorization from competent authorities.

## NXP Semiconductors

For online support resources and contact details of your local NXP office or distributor, refer to:

[www.nxp.com](http://www.nxp.com)