ZigBee Home Automation User Guide

JN-UG-3076

Revision 1.1

22 October 2013

Contents

About this Manual 13
Organisation 13
Conventions 14
Acronyms and Abbreviations 14
Related Documents 15
Support Resources 15
Trademarks 15
Chip Compatibility 15

Part I: Concept and Development Information

1. Introduction to Home Automation 19
1.1 Wireless Home Automation 19
1.2 Home Automation Benefits 20
1.3 Home Automation Application Areas 20
1.3.1 Lighting 20
1.3.2 Heating, Ventilation and Air-Conditioning (HVAC) 21
1.3.3 Shades and Window Coverings 21
1.3.4 Security Systems 21
1.4 Energy Saving 22
1.5 ZigBee Wireless Networks 22
1.6 Software Architecture 23
1.7 Interoperability and Certification 23
1.8 Commissioning 24
1.9 Internet Connectivity 24

2. Home Automation (HA) Profile 25
2.1 HA Devices 25
2.2 Common Clusters 26
2.3 Generic Devices 27
2.3.1 On/Off Switch 28
2.3.2 On/Off Output 29
2.3.3 Remote Control 30
2.3.4 Door Lock 31
2.3.5 Door Lock Controller 32
2.3.6 Simple Sensor 33

2.4 Lighting Devices 34
2.4.1 On/Off Light 35
2.4.2 Dimmable Light 36
2.4.3 Colour Dimmable Light 37
2.4.4 On/Off Light Switch 38
2.4.5 Dimmer Switch 39
2.4.6 Colour Dimmer Switch 40
2.4.7 Light Sensor 41
2.4.8 Occupancy Sensor 42

3. HA Application Development 43
3.1 Development Resources and Installation 43
3.2 HA Programming Resources 44
3.2.1 Core Resources 44
3.2.2 Cluster-specific Resources 44
3.3 Function Prefixes 45
3.4 Development Phases 45
3.5 Building an Application 46
3.5.1 Compile-Time Options 46
3.5.2 ZigBee Network Parameters 47
3.5.3 Building and Loading the Application Binary 47

4. HA Application Coding 49
4.1 HA Programming Concepts 49
4.1.1 Shared Device Structures 49
4.1.2 Addressing 51
4.1.3 OS Resources 51
4.2 Initialisation 52
4.3 Callback Functions 53
4.4 Discovering Endpoints and Clusters 54
4.5 Reading Attributes 55
4.6 Writing Attributes 57
4.7 Handling Stack and Timer Events 61
4.8 Servicing Timing Requirements 62
4.9 Time Management 62
4.9.1 Time Maintenance 63
4.9.2 Updating ZCL Time Following Sleep 64

Part II: HA Clusters

5. ZCL Clusters 67
5.1 Basic Cluster 67
5.2 Power Configuration Cluster 68
5.3 Identify Cluster 68
5.4 Groups Cluster 69
5.5 Scenes Cluster 69
5.6 On/Off Cluster 70
5.7 On/Off Switch Configuration Cluster 70
5.8 Level Control Cluster 71
5.9 Time Cluster 71
5.10 Binary Input (Basic) Cluster 71
5.11 Door Lock Cluster 72
5.12 Colour Control Cluster 72
5.13 Illuminance Measurement Cluster 73
5.14 Occupancy Sensing Cluster 73

6. Poll Control Cluster 75
6.1 Overview 75
6.2 Cluster Structure and Attributes 76
6.3 Attribute Settings 77
6.4 Poll Control Operations 78
6.4.1 Initialisation 78
6.4.2 Configuration 78
6.4.3 Operation 80
6.5 Poll Control Events 81
6.6 Functions 82
6.6.1 Server/Client Function 82
eCLD_PollControlCreatePollControl 83
6.6.2 Server Functions 85
eCLD_PollControlUpdate 86
eCLD_PollControlSetAttribute 87
6.6.3 Client Functions 88
eCLD_PollControlSetLongPollIntervalSend 89
eCLD_PollControlSetShortPollIntervalSend 91
eCLD_PollControlFastPollStopSend 93
6.7 Return Codes 94

6.8 Enumerations 94
6.8.1 ‘Attribute ID’ Enumerations 94
6.8.2 ‘Command’ Enumerations 94
6.9 Structures 95
6.9.1 tsCLD_PPCallBackMessage 95
6.9.2 tsCLD_PollControl_CheckinResponsePayload 96
6.9.3 tsCLD_PollControl_SetLongPollIntervalPayload 96
6.9.4 tsCLD_PollControl_SetShortPollIntervalPayload 97
6.9.5 tsCLD_PollControlCustomDataStructure 97
6.10 Compile-Time Options 97

7. Power Profile Cluster 101

7.1 Overview 101
7.2 Cluster Structure and Attributes 101
7.3 Power Profiles 103
7.4 Power Profile Operations 104
7.4.1 Initialisation 104
7.4.2 Adding and Removing a Power Profile (Server Only) 104
7.4.2.1 Adding a Power Profile Entry 104
7.4.2.2 Removing a Power Profile Entry 105
7.4.2.3 Obtaining a Power Profile Entry 105
7.4.3 Communicating Power Profiles 105
7.4.3.1 Requesting a Power Profile (by Client) 105
7.4.3.2 Notification of a Power Profile (by Server) 106
7.4.4 Communicating Schedule Information 106
7.4.4.1 Requesting a Schedule (by Server) 107
7.4.4.2 Notification of a Schedule (by Client) 107
7.4.4.3 Notification of Energy Phases in Power Profile Schedule (by Server) 108
7.4.4.4 Requesting the Scheduled Energy Phases (by Client) 108
7.4.5 Executing a Power Profile Schedule 108
7.4.6 Communicating Price Information 109
7.4.6.1 Requesting Cost of a Power Profile Schedule (by Server) 110
7.4.6.2 Requesting Cost of Power Profile Schedules Over a Day (by Server) 110
7.5 Power Profile Events 111
7.6 Functions 114
7.6.1 Server/Client Function 114
eCLD_PPCreatePowerProfile 115
7.6.2 Server Functions 117
eCLD_PPSchedule 118
eCLD_PPSetPowerProfileState 119
eCLD_PPAddPowerProfileEntry 120
eCLD_PPRemovePowerProfileEntry 121
eCLD_PPGetPowerProfileEntry 122
eCLD_PPPowerProfileNotificationSend 123
eCLD_PPEnergyPhaseScheduleStateNotificationSend 124
eCLD_PPPowerProfileScheduleConstraintsNotificationSend 125

eCLD_PPEnergyPhasesScheduleReqSend 127
eCLD_PPPowerProfileStateNotificationSend 128
eCLD_PPGetPowerProfilePriceSend 129
eCLD_PPGetPowerProfilePriceExtendedSend 130
eCLD_PPGetOverallSchedulePriceSend 132
7.6.3 Client Functions 133
eCLD_PPPowerProfileRequestSend 134
eCLD_PPEnergyPhasesScheduleNotificationSend 135
eCLD_PPPowerProfileStateReqSend 137
eCLD_PPEnergyPhasesScheduleStateReqSend 138
eCLD_PPPowerProfileScheduleConstraintsReqSend 139
7.7 Return Codes 141
7.8 Enumerations 141
7.8.1 ‘Attribute ID’ Enumerations 141
7.8.2 ‘Power Profile State’ Enumerations 141
7.8.3 ‘Server-Generated Command’ Enumerations 142
7.8.4 ‘Server-Received Command’ Enumerations 143
7.9 Structures 144
7.9.1 tsCLD_PPCallBackMessage 144
7.9.2 tsCLD_PPEntry 146
7.9.3 tsCLD_PP_PowerProfileReqPayload 147
7.9.4 tsCLD_PP_PowerProfilePayload 147
7.9.5 tsCLD_PP_PowerProfileStatePayload 148
7.9.6 tsCLD_PP_EnergyPhasesSchedulePayload 148
7.9.7 tsCLD_PP_PowerProfileScheduleConstraintsPayload 149
7.9.8 tsCLD_PP_GetPowerProfilePriceExtendedPayload 149
7.9.9 tsCLD_PP_GetPowerProfilePriceRspPayload 150
7.9.10 tsCLD_PP_GetOverallSchedulePriceRspPayload 150
7.9.11 tsCLD_PP_EnergyPhaseInfo 151
7.9.12 tsCLD_PP_EnergyPhaseDelay 152
7.9.13 tsCLD_PP_PowerProfileRecord 152
7.9.14 tsCLD_PPCustomDataStructure 153
7.10 Compile-Time Options 153

8. Appliance Control Cluster 155

8.1 Overview 155
8.2 Cluster Structure and Attributes 155
8.3 Sending Commands 157
8.3.1 Execution Commands from Client to Server 157
8.3.2 Status Commands from Client to Server 158
8.3.3 Status Notifications from Server to Client 158
8.4 Appliance Control Events 159

8.5 Functions 160
eCLD_ApplianceControlCreateApplianceControl 161
eCLD_ACExecutionOfCommandSend 163
eCLD_ACSignalStateSend 165
eCLD_ACSignalStateResponseORSignalStateNotificationSend 166
eCLD_ACSignalStateNotificationSend 168
eCLD_ACChangeAttributeTime 170
8.6 Return Codes 171
8.7 Enumerations 171
8.7.1 ‘Attribute ID’ Enumerations 171
8.7.2 ‘Client Command ID’ Enumerations 171
8.7.3 ‘Server Command ID’ Enumerations 172
8.8 Structures 173
8.8.1 tsCLD_ApplianceControlCallBackMessage 173
8.8.2 tsCLD_AC_ExecutionOfCommandPayload 174
8.8.3 tsCLD_AC_SignalStateResponseORSignalStateNotificationPayload 174
8.8.4 tsCLD_ApplianceControlCustomDataStructure 176
8.9 Compile-Time Options 176

9. Appliance Identification Cluster 177
9.1 Overview 177
9.2 Cluster Structure and Attributes 177
9.3 Functions 181
eCLD_ApplianceIdentificationCreateApplianceIdentification 182
9.4 Return Codes 184
9.5 Enumerations 184
9.5.1 ‘Attribute ID’ Enumerations 184
9.5.2 ‘Product Type ID’ Enumerations 184
9.6 Compile-Time Options 185

10. Appliance Events and Alerts Cluster 187
10.1 Overview 187
10.2 Cluster Structure and Attributes 187
10.3 Sending Messages 187
10.3.1 ‘Get Alerts’ Messages from Client to Server 188
10.3.2 ‘Alerts Notification’ Messages from Server to Client 188
10.3.3 ‘Event Notification’ Messages from Server to Client 189
10.4 Appliance Events and Alerts Events 189

10.5 Functions 190
eCLD_ApplianceEventsAndAlertsCreateApplianceEventsAndAlerts 191
eCLD_AEAAGetAlertsSend 193
eCLD_AEAAGetAlertsResponseORAlertsNotificationSend 194
eCLD_AEAAAlertsNotificationSend 196
eCLD_AEAAEventNotificationSend 197
10.6 Return Codes 198
10.7 Enumerations 198
10.7.1 'Command ID' Enumerations 198
10.8 Structures 199
10.8.1 tsCLD_ApplianceEventsAndAlertsCallBackMessage 199
10.8.2 tsCLD_AEAA_GetAlertsResponseORAlertsNotificationPayload 200
10.8.3 tsCLD_AEAA_EventNotificationPayload 201
10.8.4 tsCLD_ApplianceEventsAndAlertsCustomDataStructure 201
10.9 Compile-Time Options 202

11. Appliance Statistics Cluster 203
11.1 Overview 203
11.2 Cluster Structure and Attributes 204
11.3 Sending Messages 204
11.3.1 'Log Queue Request' Messages from Client to Server 205
11.3.2 'Statistics Available' Messages from Server to Client 205
11.3.3 'Log Request' Messages from Client to Server 206
11.3.4 'Log Notification' Messages from Server to Client 206
11.4 Log Operations on Server 207
11.4.1 Adding and Removing Logs 207
11.4.2 Obtaining Logs 207
11.5 Appliance Statistics Events 208
11.6 Functions 209
eCLD_ApplianceStatisticsCreateApplianceStatistics 210
eCLD_ASCAddLog 212
eCLD_ASCRemoveLog 213
eCLD_ASCGetLogsAvailable 214
eCLD_ASCGetLogEntry 215
eCLD_ASCLogQueueRequestSend 216
eCLD_ASCLogRequestSend 217
eCLD_ASCLogQueueResponseORStatisticsAvailableSend 218
eCLD_ASCStatisticsAvailableSend 220
eCLD_ASCLogNotificationORLogResponseSend 221
eCLD_ASCLogNotificationSend 223
11.7 Return Codes 224

11.8 Enumerations 224
11.8.1 'Attribute ID' Enumerations 224
11.8.2 'Client Command ID' Enumerations 224
11.8.3 'Server Command ID' Enumerations 225
11.9 Structures 225
11.9.1 tsCLD_ApplianceStatisticsCallBackMessage 225
11.9.2 tsCLD_ASC_LogRequestPayload 226
11.9.3 tsCLD_ASC_LogNotificationORLogResponsePayload 226
11.9.4 tsCLD_ASC_LogQueueResponseORStatisticsAvailablePayload 227
11.9.5 tsCLD_LogTable 227
11.9.6 tsCLD_ApplianceStatisticsCustomDataStructure 228
11.10 Compile-Time Options 228

Part III: General Reference Information

12. HA Core Functions 233
eHA_Initialise 234
eHA_Update100mS 235
eHA_RegisterOnOffSwitchEndPoint 236
eHA_RegisterOnOffOutputEndPoint 238
eHA_RegisterRemoteControlEndPoint 240
eHA_RegisterDoorLockEndPoint 242
eHA_RegisterDoorLockControllerEndPoint 244
eHA_RegisterSimpleSensorEndPoint 246
eHA_RegisterOnOffLightEndPoint 248
eHA_RegisterDimmableLightEndPoint 250
eHA_RegisterColourDimmableLightEndPoint 252
eHA_RegisterOnOffLightSwitchEndPoint 254
eHA_RegisterDimmerSwitchEndPoint 256
eHA_RegisterColourDimmerSwitchEndPoint 258
eHA_RegisterLightSensorEndPoint 260
eHA_RegisterOccupancySensorEndPoint 262

13. HA Device Structures 265
13.1 Generic Devices 265
13.1.1 tsHA_OnOffSwitchDevice 265
13.1.2 tsHA_OnOffOutputDevice 267
13.1.3 tsHA_RemoteControlDevice 269
13.1.4 tsHA_DoorLockDevice 272
13.1.5 tsHA_DoorLockControllerDevice 273
13.1.6 tsHA_SimpleSensorDevice 275

13.2 Lighting Devices 277
13.2.1 tsHA_OnOffLightDevice 277
13.2.2 tsHA_DimmableLightDevice 279
13.2.3 tsHA_ColourDimmableLightDevice 281
13.2.4 tsHA_OnOffLightSwitchDevice 283
13.2.5 tsHA_DimmerSwitchDevice 285
13.2.6 tsHA_ColourDimmerSwitchDevice 288
13.2.7 tsHA_LightSensorDevice 290
13.2.8 tsHA_OccupancySensorDevice 291

Part IV: Appendices

A. Custom Endpoints 295
A.1 HA Devices and Endpoints 295
A.2 Cluster Creation Functions 296
A.3 Custom Endpoint Set-up 296
A.3.1 Custom Endpoint Structure 297

About this Manual

This manual provides an introduction to the ZigBee Home Automation (HA) application profile and describes the use of the NXP HA Application Programming Interface (API) for the JN5168 wireless microcontroller. The manual contains both operational and reference information relating to the HA API, including descriptions of the C functions and associated resources (e.g. structures).

Note: Many clusters used by the devices in the HA profile are from the ZigBee Cluster Library (ZCL). These clusters are fully detailed in the *ZCL User Guide (JN-UG-3077),* available from the NXP Wireless Connectivity TechZone (see “Support Resources” on page 15).

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

- Information on ZigBee PRO wireless networks is provided in the *ZigBee PRO Stack User Guide (JN-UG-3048)*, available from NXP.
- The ZigBee HA profile is defined in the *ZigBee Home Automation Profile Specification (053520)*, available from the ZigBee Alliance at **www.zigbee.org**.

Organisation

This manual is divided into four parts:

- Part I: Concept and Development Information comprises four chapters:
  - Chapter 1 introduces the principles of Home Automation (HA)
  - Chapter 2 describes the devices available in the ZigBee HA application profile
  - Chapter 3 provides an overview of HA application development
  - Chapter 4 describes the essential aspects of coding an HA application
- Part II: HA Clusters comprises seven chapters:
  - Chapter 5 outlines the clusters from the ZigBee Cluster Library (ZCL) that are used in the HA profile
  - Chapter 6 describes the Poll Control cluster of the HA profile
  - Chapter 7 describes the Power Profile cluster of the HA profile
  - Chapter 8 describes the Appliance Control cluster of the HA profile
  - Chapter 9 describes the Appliance Identification cluster of the HA profile
  - Chapter 10 describes the Appliance Events and Alerts cluster of the HA profile

    - Chapter 11 describes the Appliance Statistics cluster of the HA profile
- Part III: General Reference Information comprises two chapters:
    - Chapter 12 details the core functions of the HA API, including initialisation and device registration functions
    - Chapter 13 details the device structures included in the HA API
- Part IV: Appendices contains an appendix which describes how to set up custom endpoints.

Conventions

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

Function parameters are represented in *italics* type.

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

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

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

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

Acronyms and Abbreviations

APDU	Application Protocol Data Unit
API	Application Programming Interface
HA	Home Automation
SDK	Software Developer’s Kit
ZCL	ZigBee Cluster Library

Related Documents

JN-UG-3048	ZigBee PRO Stack User Guide
JN-UG-3077	ZigBee Cluster Library User Guide
JN-UG-3075	JenOS User Guide
JN-UG-3064	SDK Installation and User Guide
JN-UG-3007	JN51xx Flash Programmer User Guide
JN-AN-1189	ZigBee Home Automation Demonstration Application Note
053520	ZigBee Home Automation Profile Specification [from ZigBee Alliance]
075123	ZigBee Cluster Library Specification [from ZigBee Alliance]
BS EN 50523	Household appliances interworking [from British Standards Institute]

Support Resources

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

www.nxp.com/techzones/wireless-connectivity

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

Trademarks

All trademarks are the property of their respective owners.

Chip Compatibility

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

JN516x (currently only JN5168-001)

Part I: Concept and Development Information

1. Introduction to Home Automation

Home automation is not new! Throughout history, we have continuously strived to automate tasks in the home in order to make our lives easier. Technology has now advanced to the point at which we wish to take an integrated approach to home automation, allowing appliances to communicate with each other and to be controlled in flexible ways. A wireless network approach to this communication and control provides an easy, cost-effective and scalable solution to home automation.

1.1 Wireless Home Automation

A network approach to home automation allows a diverse range of potential applications, including:

- Lighting
- Heating and cooling
- Shades, blinds and curtains
- Home security

Possible application areas of home automation are described in Section 1.3.

Multiple home automation applications can be controlled through the same network infrastructure. However, the installation of a wired home automation network is costly and disruptive unless carried out during the construction or refurbishment of the building. The advantages of a radio-based wireless home automation network are:

- No expensive and disruptive network wiring to be installed in the building
- Can be easily and cheaply installed at any time with minimal disruption
- Can be expanded, as required, at any time to cover a wider physical area
- Can be scaled, as required, at any time to incorporate more application areas

The ZigBee Home Automation (HA) application profile, described in this manual, facilitates this wireless networking solution.

Note: Not all of the application areas covered by the ZigBee Home Automation profile are currently supported by the HA profile from NXP - see Section 1.3.

1.2 Home Automation Benefits

Home automation brings a variety of benefits, depending on the application area(s). These potential benefits include:

- Easier lifestyle
- Convenience of flexible control and remote control
- Increased safety around the home
- Improved security of the home
- Energy savings with associated cost savings and environmental benefits

The energy saving features of home automation are outlined in Section 1.4.

1.3 Home Automation Application Areas

Home automation solutions can be applied to many aspects of the home, as described in the sub-sections below.

Note: Not all of the application areas described below are currently supported by the ZigBee Home Automation (HA) profile from NXP.

1.3.1 Lighting

Lighting systems can be implemented with the following functionality:

- Control lights from various points, including wall-switches, occupancy sensors, remote control units, smartphones, tablets and computers
- Control lights in terms of brightness and colour (for colour lamps)
- Control a pre-defined group of lights by a single action
- Definition of brightness and/or colour settings for one or more lights, forming a ‘scene’ for mood lighting

Lighting solutions are supported by NXP’s ZigBee HA profile.

Note: For a pure lighting system (with no other HA application areas), the ZigBee Light Link (ZLL) profile provides an alternative to the Home Automation profile. For details of NXP’s ZLL profile, refer to the *ZigBee Light Link User Guide (JN-UG-3091)*.

1.3.2 Heating, Ventilation and Air-Conditioning (HVAC)

HVAC systems can be implemented with the following functionality:

- Control heating and/or air-conditioning from various points, including wall-mounted control units, thermostats, occupancy sensors, remote control units, smartphones, tablets and computers
- Control the heating and/or air-conditioning in individual rooms according to their use and/or occupancy
- Control a pre-defined group of heaters or fans by a single action
- Definition of heating/cooling settings (e.g. temperatures) for one or more rooms, forming a 'scene'

HVAC solutions are not currently supported by NXP's ZigBee HA profile.

1.3.3 Shades and Window Coverings

The control of shades and window coverings (blinds and curtains) can be implemented with the following functionality:

- Control shades and window coverings from various points, including wall-mounted control units, remote control units, smartphones, tablets and computers
- Open/close shades and window coverings, including partial opening/closing
- Control a pre-defined group of shades or window coverings by a single action
- Definition of open/close settings for one or more shades or window coverings, forming a 'scene'

Shade and window covering solutions are not currently supported by NXP's ZigBee HA profile.

1.3.4 Security Systems

Security systems (intruder and fire) can be implemented with the following functionality:

- Control the security system from various points, including wall-mounted control units, remote control units, smartphones, tablets and computer
- Control a pre-defined group of security sensors or door locks by a single action
- Definition of security settings for one or more sensors or door-locks, forming a 'scene'

Security systems are not currently supported by NXP's ZigBee HA profile, except door locks which are supported.

1.4 Energy Saving

A ZigBee Home Automation 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 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 appliances, such as lights, only when a person is detected (and switch off when a person is no longer detected). As an example, this method may be very useful for controlling lights in a corridor or garage, or outside lights.
- **Energy monitoring:** The power consumption of an HA system may be monitored.

1.5 ZigBee Wireless Networks

ZigBee Home Automation (HA) is a public application profile that has been devised by the ZigBee Alliance to support home automation solutions based on the ZigBee PRO wireless network protocol. ZigBee PRO is fully described in the *ZigBee PRO Stack User Guide (JN-UG-3048)*.

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

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

Note: The software architecture for HA, in terms of a protocol stack, is described in more detail in Section 1.6.

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

1.6 Software Architecture

The ZigBee Home Automation profile operates in conjunction with the ZigBee PRO wireless network protocol. The software stack which runs on each HA node is illustrated in Figure 1 below.

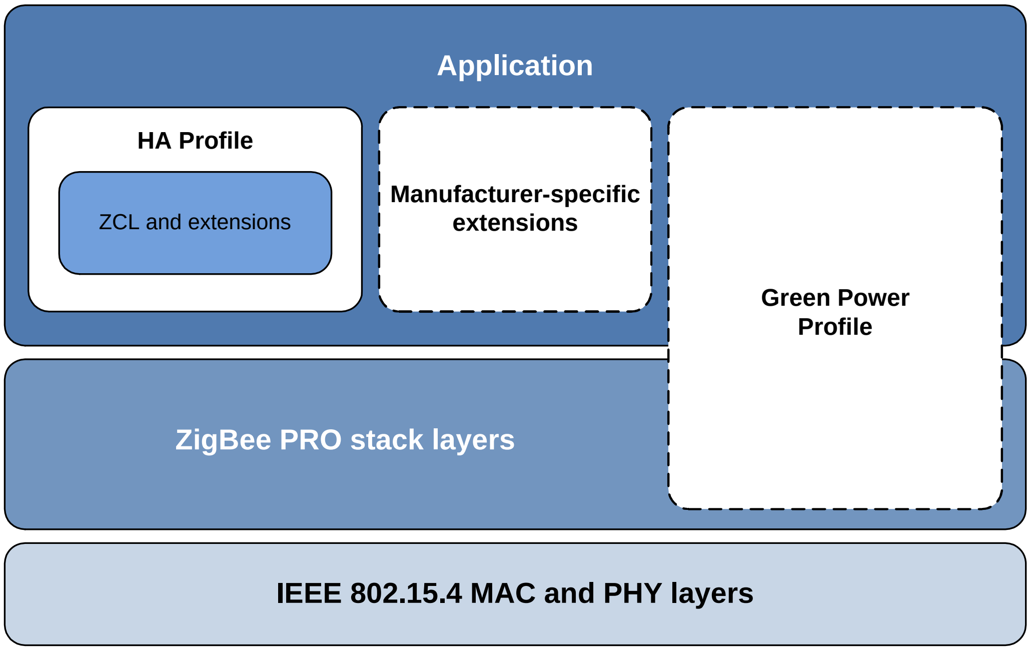


Figure 1: HA Software Stack

The main features of the above stack are as follows:

- Manufacturer application, which interfaces to the underlying ZigBee PRO stack layers and controls the appliance hardware of the node, and uses:
  - HA profile, including ZCL resources (ZCL clusters and extensions)
  - Optional manufacturer-specific extensions to the HA profile
  - Optional ZigBee Green Power profile
- ZigBee PRO stack layers, as described in the *ZigBee PRO Stack User Guide (JN-UG-3048)*

1.7 Interoperability and Certification

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

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

In addition, the HA profile is designed to be interoperable at the network layer with other public ZigBee application profiles.

1.8 Commissioning

The process of introducing an HA device into an HA network is called commissioning. This involves finding an HA network, joining the network and ultimately binding an endpoint on the new device to a compatible endpoint on an existing device, in order to allow the new device to perform its function within the network (e.g. pairing a new light-switch with an existing lamp so that the switch can control the lamp).

The HA software solution from NXP supports EZ-mode commissioning (defined in the Home Automation Specification 1.2). It is a ZigBee requirement that all HA devices support this mode of commissioning (except a ‘Commissioning Director’).

In EZ-mode commissioning, an HA device is commissioned by means of user interactions, such as button-presses. This commissioning mode does allow some automatic behaviour, such as automatically joining a network at power-up, but some user intervention will always be required to complete the commissioning process.

An EZ-mode Commissioning module is provided in the NXP HA software and is described in the *ZigBee Cluster Library User Guide (JN-UG-3077)*.

Note: ZigBee specify the commissioning terminology that should be used by all HA product documentation in order to ensure consistency between products and manufacturers. This recommended terminology is also detailed in the *ZCL User Guide (JN-UG-3077).*

1.9 Internet Connectivity

ZigBee Home Automation offers the possibility of controlling the appliances in an HA system via the Internet. Thus, this control can be performed from any Internet-connected device (PC, tablet, smartphone) located anywhere in the World (e.g. while on holiday in another country).

Access from the Internet requires the HA 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 an HA system over the Internet, the system could also be configured from a device on the Internet (e.g. groups, scenes and timers).

2. Home Automation (HA) Profile

Home Automation (HA) is ZigBee application profile 0x0104. This chapter details the ZigBee devices available in the HA profile and the clusters that they use.

Note: This manual assumes that you are already familiar with ZigBee PRO concepts such as endpoints, profiles, clusters and attributes. For more information, refer to the *ZigBee PRO Stack User Guide (JN-UG-3048)*, available from the NXP Wireless Connectivity TechZone.

2.1 HA Devices

This manual covers the following devices from the ZigBee Home Automation application profile, which are divided into application-oriented categories:

- **Generic devices** (described in Section 2.3)
  - On/Off Switch
  - On/Off Output
  - Remote Control
  - Door Lock
  - Door Lock Controller
  - Simple Sensor
- **Lighting devices** (described in Section 2.4)
  - On/Off Light
  - Dimmable Light
  - Colour Dimmable Light
  - On/Off Light Switch
  - Dimmer Switch
  - Colour Dimmer Switch
  - Light Sensor
  - Occupancy Sensor

The HA profile contains many other devices that are not currently implemented in the NXP HA software - for the full list of HA devices, refer to the *ZigBee Home Automation Profile Specification (053520)*, available from the ZigBee Alliance (**www.zigbee.org**).

2.2 Common Clusters

The HA devices are defined by the clusters that they use. Some clusters are common to most HA devices - these are detailed in the table below.

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

Server (Input) Side	Client (Output) Side
Mandatory	
Basic	
Identify	
Optional	
Clusters with reporting capability	Clusters with reporting capability
Power Configuration	Time
Device Temperature Configuration	OTA Bootload
Alarms	Partition
Electrical Measurement	
Poll Control	
Partition	
Manufacturer-specific	Manufacturer-specific

Table 1: Common Clusters for HA Devices

2.3 Generic Devices

This section details the HA Generic Devices, including the clusters that they support. The Generic Devices are listed in the table below along with their Device IDs and references to the sub-sections in which they are described.

Generic Device	Device ID	Reference
On/Off Switch	0x0000	Section 2.3.1
On/Off Output	0x0002	Section 2.3.2
Remote Control	0x0006	Section 2.3.3
Door Lock	0x000A	Section 2.3.4
Door Lock Controller	0x000B	Section 2.3.5
Simple Sensor	0x000C	Section 2.3.6

Table 2: Generic Devices

Note: The clusters used by these devices are contained in the ZigBee Cluster Library and are described in the *ZCL User Guide (JN-UG-3077)*, available from the NXP Wireless Connectivity TechZone. However, not all the listed clusters are currently supported by the NXP software.

2.3.1 On/Off Switch

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

Note: This device should be used only when a more specific device profile is not available - for example, the On/Off Light Switch device should be used to control the On/Off Light device.

- The Device ID is 0x0000
- The header file for the device is **on_off_switch.h**
- The device structure, `tsHA_OnOffSwitchDevice`, is listed in Section 13.1.1
- The endpoint registration function for the device, **eHA_RegisterOnOffSwitchEndPoint()**, is detailed in Chapter 12

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

Server (Input) Side	Client (Output) Side
Mandatory	
Basic	On/Off (subject to binding)
Identify	Identify
Optional	
See Table 1 on page 26	See Table 1 on page 26
On/Off Switch Configuration	Scenes
	Groups

Table 3: Clusters for On/Off Switch

2.3.2 On/Off Output

The On/Off Output device is capable of being switched on and off.

Note: This device should be used only when a more specific device profile is not available - for example, the On/Off Light device.

- The Device ID is 0x0002
- The header file for the device is **on_off_output.h**
- The device structure, `tsHA_OnOffOutputDevice`, is listed in Section 13.1.2
- The endpoint registration function for the device, **eHA_RegisterOnOffOutputEndPoint()**, is detailed in Chapter 12

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

Server (Input) Side	Client (Output) Side
Mandatory	
Basic	
Identify	
On/Off	
Scenes	
Groups	
Optional	
See Table 1 on page 26	See Table 1 on page 26

Table 4: Clusters for On/Off Output

2.3.3 Remote Control

The Remote Control device is used to control and monitor one or more other devices. The client side is typically incorporated in a handheld unit, with the server side in the node(s) to be controlled/monitored.

- The Device ID is 0x0006
- The header file for the device is **remote_control.h**
- The device structure, `tsHA_RemoteControlDevice`, is listed in Section 13.1.3
- The endpoint registration function for the device, **eHA_RegisterRemoteControlEndPoint()**, is detailed in Chapter 12

The clusters used by the Remote Control device are listed in the table below.

Server (Input) Side	**Client (Output) Side**
Mandatory	
Basic	At least one optional cluster
Identify	
Optional	
See Table 1 on page 26	See Table 1 on page 26
	Basic
	Identify
	On/Off
	Level Control
	Groups
	Scenes
	Colour Control
	Pump Configuration and Control
	Shade Configuration
	On/Off Switch Configuration
	Temperature Measurement
	Illuminance Level Sensing
	Illuminance Measurement
	Window Covering
	Door Lock
	Thermostat

Table 5: Clusters for Remote Control

2.3.4 Door Lock

The Door Lock device is able to receive commands from a Door Lock Controller device (see Section 2.3.5).

- The Device ID is 0x000A
- The header file for the device is **door_lock.h**
- The device structure, `tsHA_DoorLockDevice`, is listed in Section 13.1.4
- The endpoint registration function for the device, **eHA_RegisterDoorLockEndPoint()**, is detailed in Chapter 12

The clusters used by the Door Lock device are listed in the table below.

Server (Input) Side	Client (Output) Side
Mandatory	
Basic	
Identify	
Door Lock	
Scenes	
Groups	
Optional	
See Table 1 on page 26	See Table 1 on page 26
Alarms	Time
Power Configuration	OTA Bootload
Poll Control	

Table 6: Clusters for Door Lock

Note: In Home Automation, the Door Lock cluster is enhanced to allow Application-level security to be used (in addition to the default Network-level security). For details, refer to the *ZCL User Guide (JN-UG-3077)*.

2.3.5 Door Lock Controller

The Door Lock Controller device is able to send commands to a Door Lock device (see Section 2.3.4).

- The Device ID is 0x000B
- The header file for the device is **door_lock_controller.h**
- The device structure, `tsHA_DoorLockControllerDevice`, is listed in Section 13.1.5
- The endpoint registration function for the device, **eHA_RegisterDoorLockControllerEndPoint()**, is detailed in Chapter 12

The clusters used by the Door Lock Controller device are listed in the table below.

Server (Input) Side	Client (Output) Side
Mandatory	
Basic	Door Lock
Identify	Scenes
	Group
	Identify
Optional	
See Table 1 on page 26	See Table 1 on page 26

Table 7: Clusters for Door Lock Controller

Note: In Home Automation, the Door Lock cluster is enhanced to allow Application-level security to be used (in addition to the default Network-level security). For details, refer to the *ZCL User Guide (JN-UG-3077)*.

2.3.6 Simple Sensor

The Simple Sensor device is able to accept a binary input from an on/off device such as magnetic window contacts.

- The Device ID is 0x000C
- The header file for the device is **simple_sensor.h**
- The device structure, `tsHA_SimpleSensorDevice`, is listed in Section 13.1.6
- The endpoint registration function for the device, **eHA_RegisterSimpleSensorEndPoint()**, is detailed in Chapter 12

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

Server (Input) Side	Client (Output) Side
Mandatory	
Basic	Identify
Identify	
Binary Input (Basic)	
Optional	
See Table 1 on page 26	See Table 1 on page 26

Table 8: Clusters for Simple Sensor

2.4 Lighting Devices

This section details the Lighting Devices, including the clusters that they support. The Lighting Devices are listed in the table below along with their Device IDs and references to the sub-sections in which they are described.

Lighting Device	Device ID	Reference
On/Off Light	0x0100	Section 2.4.1
Dimmable Light	0x0101	Section 2.4.2
Colour Dimmable Light	0x0102	Section 2.4.3
On/Off Light Switch	0x0103	Section 2.4.4
Dimmer Switch	0x0104	Section 2.4.5
Colour Dimmer Switch	0x0105	Section 2.4.6
Light Sensor	0x0106	Section 2.4.7
Occupancy Sensor	0x0107	Section 2.4.8

Table 9: Lighting Devices

The possible pairings of these devices are summarised in the table below:

Controller Device	Controlled Device	Description
On/Off Light Switch	On/Off Light	Switch or sensor puts light in one of two states, on or off
Light Sensor		
Occupancy Sensor		
Dimmer Switch	Dimmable Light	Switch or sensor controls luminance of light between maximum and minimum levels, or puts light in on or off state
Light Sensor		
Occupancy Sensor		
Colour Dimmer Switch	Colour Dimmable Light	Switch or sensor controls hue, saturation and luminance of multi-colour light, or puts light in on or off state
Light Sensor		
Occupancy Sensor		

Table 10: Pairings of Lighting Devices

Note: The clusters used by these devices are contained in the ZigBee Cluster Library and are described in the *ZCL User Guide (JN-UG-3077)*, available from the NXP Wireless Connectivity TechZone. However, not all the listed clusters are currently supported by the NXP software.

2.4.1 On/Off Light

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

- The Device ID is 0x0100
- The header file for the device is **on_off_light.h**
- The device structure, `tsHA_OnOffLightDevice`, is listed in Section 13.2.1
- The endpoint registration function for the device, **eHA_RegisterOnOffLightEndPoint()**, is detailed in Chapter 12

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

Server (Input) Side	Client (Output) Side
Mandatory	
Basic	
Identify	
On/Off	
Scenes	
Groups	
Optional	
See Table 1 on page 26	See Table 1 on page 26
	Occupancy Sensing

Table 11: Clusters for On/Off Light

2.4.2 Dimmable Light

The Dimmable Light device is a light that can have its luminance varied, and can be switched on and off.

- The Device ID is 0x0101
- The header file for the device is **dimmable_light.h**
- The device structure, `tsHA_DimmableLightDevice`, is listed in Section 2.4.2
- The endpoint registration function for the device, **eHA_RegisterDimmableLightEndPoint()**, is detailed in Chapter 12

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

Server (Input) Side	Client (Output) Side
Mandatory	
Basic	
Identify	
On/Off	
Level Control	
Scenes	
Groups	
Optional	
See Table 1 on page 26	See Table 1 on page 26
	Occupancy Sensing

Table 12: Clusters for Dimmable Light

2.4.3 Colour Dimmable Light

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

- The Device ID is 0x0102
- The header file for the device is **colour_dimmable_light.h**
- The device structure, `tsHA_ColourDimmableLightDevice`, is listed in Section 2.4.3
- The endpoint registration function for the device, **eHA_RegisterColourDimmableLightEndPoint()**, is detailed in Chapter 12

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

Server (Input) Side	Client (Output) Side
Mandatory	
Basic	
Identify	
On/Off	
Level Control	
Colour Control	
Scenes	
Groups	
Optional	
See Table 1 on page 26	See Table 1 on page 26
	Occupancy Sensing

Table 13: Clusters for Colour Dimmable Light

2.4.4 On/Off Light Switch

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

- The Device ID is 0x0103
- The header file for the device is **on_off_light_switch.h**
- The device structure, `tsHA_OnOffLightSwitchDevice`, is listed in Section 13.2.4
- The endpoint registration function for the device, **eHA_RegisterOnOffLightSwitchEndPoint()**, is detailed in Chapter 12

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

Server (Input) Side	**Client (Output) Side**
Mandatory	
Basic	On/Off
Identify	Identify
Optional	
See Table 1 on page 26	See Table 1 on page 26
On/Off Switch Configuration	Scenes
	Groups

Table 14: Clusters for On/Off Light Switch

Note: The On/Off Light Switch supports the same clusters as the On/Off Switch (see Section 2.3.1) and has the same functionality.

2.4.5 Dimmer Switch

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

- The Device ID is 0x0104
- The header file for the device is **dimmer_switch.h**
- The device structure, `tsHA_DimmerSwitchDevice`, is listed in Section 13.2.5
- The endpoint registration function for the device, **eHA_RegisterDimmerSwitchEndPoint()**, is detailed in Chapter 12

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

Server (Input) Side	Client (Output) Side
Mandatory	
Basic	On/Off
Identify	Identify
	Level Control
Optional	
See Table 1 on page 26	See Table 1 on page 26
On/Off Switch Configuration	Scenes
	Groups

Table 15: Clusters for Dimmer Switch

Note: The Dimmer Switch supports the same clusters as the Level Control Switch (see Section 2.3.2) and has the same functionality.

2.4.6 Colour Dimmer Switch

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

- The Device ID is 0x0105
- The header file for the device is **colour_dimmer_switch.h**
- The device structure, `tsHA_ColourDimmerSwitchDevice`, is listed in Section 13.2.6
- The endpoint registration function for the device, **eHA_RegisterColourDimmerSwitchEndPoint()**, is detailed in Chapter 12

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

Server (Input) Side	**Client (Output) Side**
Mandatory	
Basic	On/Off
Identify	Level Control
	Colour Control
	Identify
Optional	
See Table 1 on page 26	See Table 1 on page 26
On/Off Switch Configuration	Scenes
	Groups

Table 16: Clusters for Colour Dimmer Switch

2.4.7 Light Sensor

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

- The Device ID is 0x0106
- The header file for the device is **light_sensor.h**
- The device structure, `tsHA_LightSensorDevice`, is listed in Section 13.2.7
- The endpoint registration function for the device, **eHA_RegisterLightSensorEndPoint()**, is detailed in Chapter 12

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

Server (Input) Side	**Client (Output) Side**
Mandatory	
Basic	Identify
Identify	
Illuminance Measurement	
Optional	
See Table 1 on page 26	See Table 1 on page 26
	Groups

Table 17: Clusters for Light Sensor

2.4.8 Occupancy Sensor

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

- The Device ID is 0x0107
- The header file for the device is **occupancy_sensor.h**
- The device structure, `tsHA_OccupancySensorDevice`, is listed in Section 2.4.8
- The endpoint registration function for the device, **eHA_RegisterOccupancySensorEndPoint()**, is detailed in Chapter 12

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

Server (Input) Side	Client (Output) Side
Mandatory	
Basic	Identify
Identify	
Occupancy Sensing	
Optional	
See Table 1 on page 26	See Table 1 on page 26
	Groups

Table 18: Clusters for Occupancy Sensor

3. HA Application Development

This chapter provides basic guidance on developing a ZigBee Home Automation (HA) application. The topics covered in this chapter include:

- Development resources and their installation (Section 3.1)
- HA 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 HA applications for the JN5168 wireless microcontroller. An HA application is developed as a ZigBee PRO application that uses the NXP ZigBee PRO APIs in conjunction with JenOS (Jennic Operating System), together with HA-specific and ZCL resources. All resources are available from the NXP Wireless Connectivity TechZone (see "Support Resources" on page 15) 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 SDK (JN-SW-4067):** This installer contains the ZigBee PRO stack and HA profile software, including a number of C APIs:
  - HA 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.
- **SDK Toolchain (JN-SW-4041):** This installer contains the tools that you will use in creating an application, including:
  - Eclipse IDE (Integrated Development Environment)
  - JN51xx compiler
  - JN51xx Flash Programmer
  - Cygwin Command Line Interface (CLI)

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

An HA demonstration application is provided in the Application Note *ZigBee Home Automation Demonstration (JN-AN-1189)*, available from NXP.

3.2 HA Programming Resources

The NXP HA API contains a range of resources (such as functions and structures), 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 HA profile handle the basic operations required in an HA network, irrespective of the clusters used. Some of these resources are provided in the HA API and some are provided in the ZCL API.

- Functions for the following operations are provided in the HA API and are detailed in Chapter 12:
  - Initialising the HA API (one function)
  - Servicing timing requirements (one function)
  - Registering a device endpoint on an HA node (one function per device)
- Functions for the following operations are provided in the ZCL API and are detailed in the *ZCL User Guide (JN-UG-3077)*:
  - 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 an HA device

Use of the above functions is described in Chapter 4.

3.2.2 Cluster-specific Resources

An HA device uses certain mandatory and optional ZigBee clusters, as listed for each device in Chapter 2.

Many of these clusters are taken from the ZCL and introduced in Chapter 5. They are fully described in the *ZigBee Cluster Library User Guide (JN-UG-3077)*.

3.3 Function Prefixes

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

- **HA functions:** Used to interact with the HA profile and prefixed with **xHA_**
- **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 HA specification, they are prefixed with **xHA_**
  - 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 HA-specific are detailed in this manual. Functions which relate to clusters of the ZCL are detailed in the *ZCL User Guide (JN-UG-3077)*.

3.4 Development Phases

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

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

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

3.5 Building an Application

This section outlines how to build an HA application developed for the JN5168 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 HA 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 Home Automation Demonstration (JN-AN-1189)*, which can be used as a template.

Number of Endpoints

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

```
#define HA_NUMBER_OF_ENDPOINTS   3
```

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

Server and Client Options

Many clusters used in HA 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: HA Clusters. For clusters from the ZCL, refer to the *ZCL User Guide (JN-UG-3077)*.

3.5.2 ZigBee Network Parameters

HA 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-3048)*.

3.5.3 Building and Loading the Application Binary

An HA application for the JN5168 device is built like any other ZigBee PRO application. The build is normally carried out using the Eclipse IDE. This is the method that we recommend, although it is also possible to use makefiles directly from the command line (Cygwin).

For instructions on building an application in the Eclipse IDE, refer to the *SDK Installation and User Guide (JN-UG-3064)*. This guide also indicates how to load the built application binary file into a JN5168-based node using the JN51xx Flash Programmer launched from within Eclipse. Alternatively, you can use the JN51xx Flash Programmer directly. In either case, you will need to refer to the *JN51xx Flash Programmer User Guide (JN-UG-3007)* as part of this procedure.

4. HA Application Coding

This chapter covers general aspects of HA application coding, including essential HA 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-3077)*.

4.1 HA Programming Concepts

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

4.1.1 Shared Device Structures

In each HA device, attribute values are exchanged between the application and the HA library by means of a shared structure. This structure is protected by a mutex (described in the *ZCL User Guide (JN-UG-3077)*). The structure for a particular HA device contains structures for the clusters supported by that device (see Chapter 2). The available device structures are provided in Chapter 13.

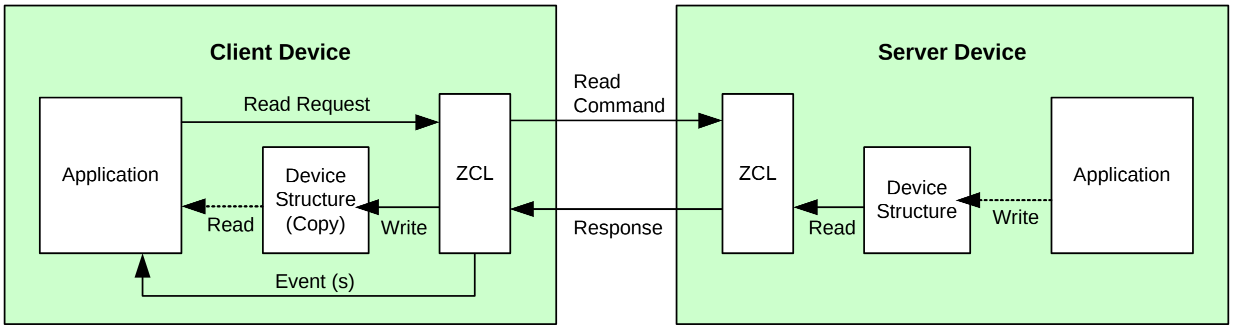
Note: In order to use a cluster which is supported by a device, the relevant option for the cluster must be specified at build-time - see Section 3.5.3.

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 1 below. For more detailed descriptions of these operations, refer to Section 4.5 and Section 4.6.

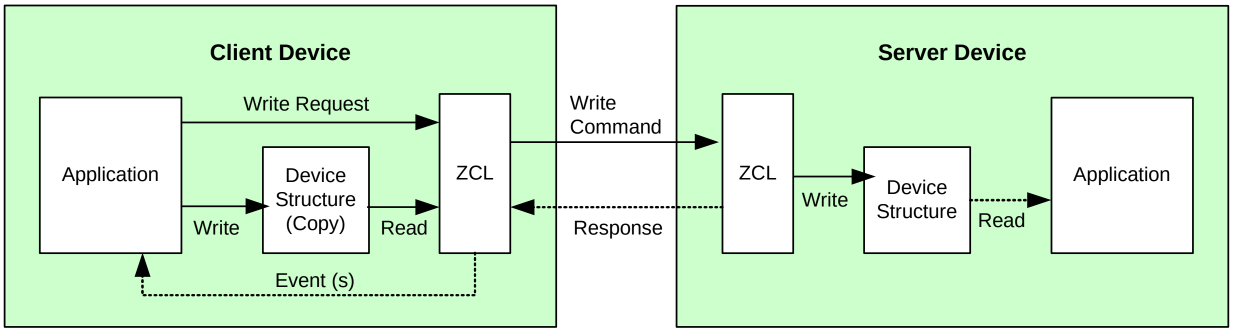
Reading Remote Attributes



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

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

Writing Remote Attributes



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

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

Figure 1: Operations using Shared Device Structure

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

4.1.2 Addressing

Communications between devices in an HA 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 HA 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-3077)*. Enumerations are provided for the addressing mode and broadcast mode in this structure, and are also detailed in the above manual.

4.1.3 OS Resources

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

The JenOS resources for an application are allocated using the JenOS Configuration Editor, which is provided as an NXP-specific plug-in for the Eclipse IDE. Use of the JenOS Configuration Editor for an HA application should be based on the HA demonstration application (rather than on the standard ZigBee PRO stack template) to ensure that the extra JenOS resources required by the HA 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-3077)*.

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

The task that the HA library and ZCL use to process incoming messages is defined in the HA demonstration application. Callbacks from the HA library and ZCL to the application will be in the context of this task. The HA 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 an HA 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

An HA application is initialised like a normal ZigBee PRO application, as described in the section “Forming a Network” of the *ZigBee PRO Stack User Guide (JN-UG-3048)*, except there is no need to explicitly start the ZigBee PRO stack using the function **ZPS_eAplZdoStartStack()**. In addition, some HA initialisation must be performed in the application code.

Initialisation of an HA application must be performed in the following places and order:

1. In the header file **zcl_options.h**, enable the required compile-time options. These options include the clusters to be used by the device, the client/server status of each cluster and the optional attributes for each cluster. For more information on compile-time options, refer to Section 3.5.1.
2. In the application, create an instance of the device structure by declaring a file scope variable - for example:

   ```
   tsHA_DimmableLightDevice sDevice;
   ```
3. In the initialisation part of the application, set up the HA device(s) handled by your code, as follows:

   a) Set the initial values of the cluster attributes to be used by the device - for example:

   ```
   sDevice.sBasicCluster.u8StackVersion = 1;
   sDevice.sBasicCluster....
   ```

   These settings should appear in the code after JenOS has been started and before the HA initialisation function is called (next step).

   b) After calling **ZPS_eAplAfInit()**, call the HA initialisation function, **eHA_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.

c) Register each device by calling the relevant device registration function - for example, **eHA_RegisterDimmableLightEndPoint()**. In this function call, the device must be allocated a unique endpoint (in the range 1-240). In addition, its device structure must be specified as well as a user-defined callback function that will be invoked by the HA library when an event occurs relating to the endpoint (see Section 4.3). As soon as this function has been called, the shared device structure can be read by another device.

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

Note: The set of endpoint registration functions for the different HA device types are detailed in Chapter 12.

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 HA library when the endpoint is registered using the registration function for the HA device type that the endpoint supports - for example, using **eHA_RegisterOnOffLightEndPoint()** for an On/Off Light device (see Chapter 12).
- **General Callback Function:** Events that do not have an associated endpoint are delivered via a callback function that is registered with the HA library through the function **eHA_Initialise()**. For example, stack leave and join events can be received by the application through this callback function.

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

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

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

4.4 Discovering Endpoints and Clusters

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

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

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

Once a relevant node and endpoint have been identified:

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

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

4.5 Reading Attributes

Attributes can be read using a general ZCL function, or using an HA 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-3077)*. 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 2. Note that instances of the shared device structure (which contains the relevant attributes) exist on both the source and destination nodes. The events generated from a 'read attributes' request are further described in 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-3077)*
3. Reads the relevant attribute values from the shared device structure and creates a 'read attributes' response message containing the read values.
4. Generates an E_ZCL_CBET_UNLOCK_MUTEX event for the endpoint callback function, which should now unlock the mutex that protects the shared device structure (other application tasks can now access the structure).
5. Sends the 'read attributes' response to the source node of the request.

3. On Source Node (Client)

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

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

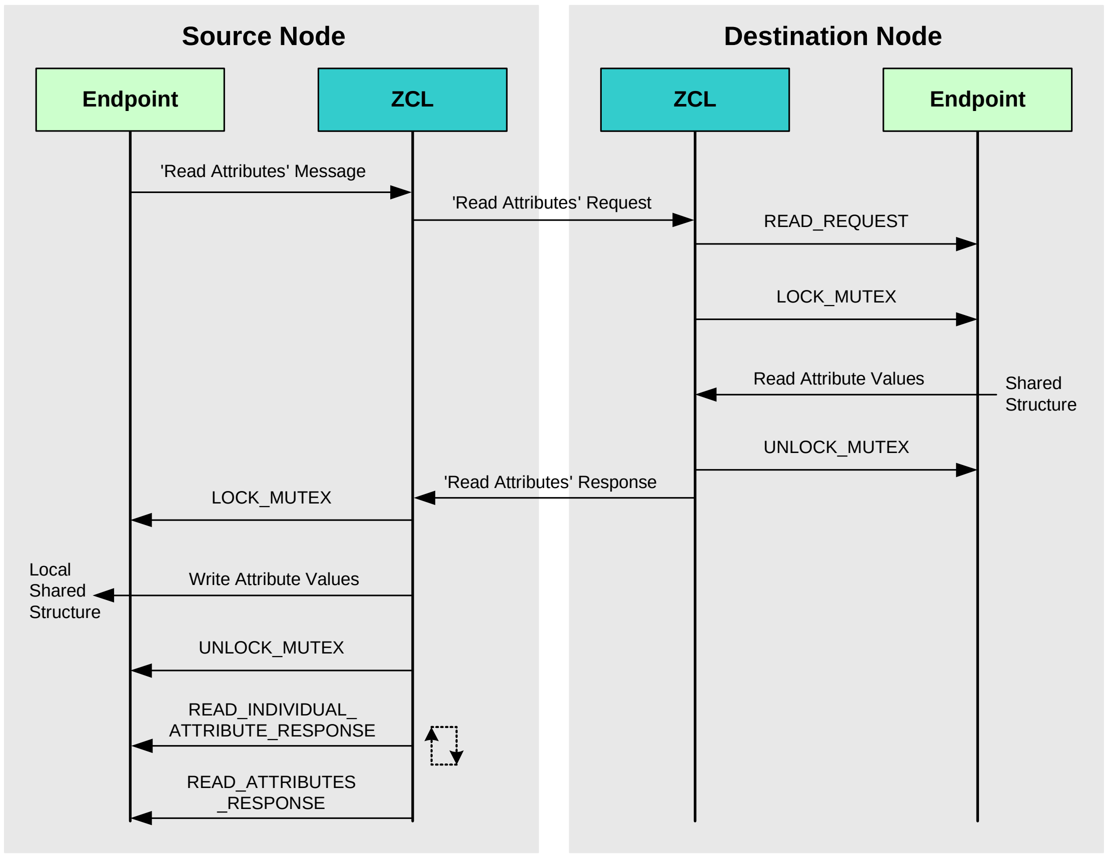


Figure 2: ‘Read Attributes’ Request and Response

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

4.6 Writing Attributes

The ability to write attribute values to a remote cluster is required by some HA 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 3. Note that instances of the shared device structure (which contains the relevant attributes) must be maintained on both the source and destination nodes. The events generated from a 'write attributes' request are further described in Section 4.7.

1. On Source Node (Client)

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

1. Locks the mutex that protects the local instance of the shared device structure that contains the attributes to be updated - for information on mutexes, refer to the *ZCL User Guide (JN-UG-3077).*
2. Writes one or more updated attribute values to the local instance of the shared device structure.
3. Unlocks the mutex that protects the local instance of the shared device structure.
4. Calls one of the above ZCL 'write attributes' functions to submit a request to update the relevant attributes on a cluster on a remote node. The information required by this function includes the following:
   - Source endpoint (from which the write request is to be sent)
   - Address of destination node for request
   - Destination endpoint (on destination node)
   - Identifier of the cluster containing the attributes [enumerations provided]
   - Number of attributes to be written
   - Array of identifiers of attributes to be written [enumerations provided]

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

2. On Destination Node (Server)

On receiving the 'write attributes' request, the 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-3077).*

3. Writes the relevant attribute values to the shared device structure - an E_ZCL_CBET_WRITE_INDIVIDUAL_ATTRIBUTE event is generated for each individual attempt to write an attribute value, which the endpoint callback function can use to keep track of the successful and unsuccessful writes.

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

3. On Source Node (Client)

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

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

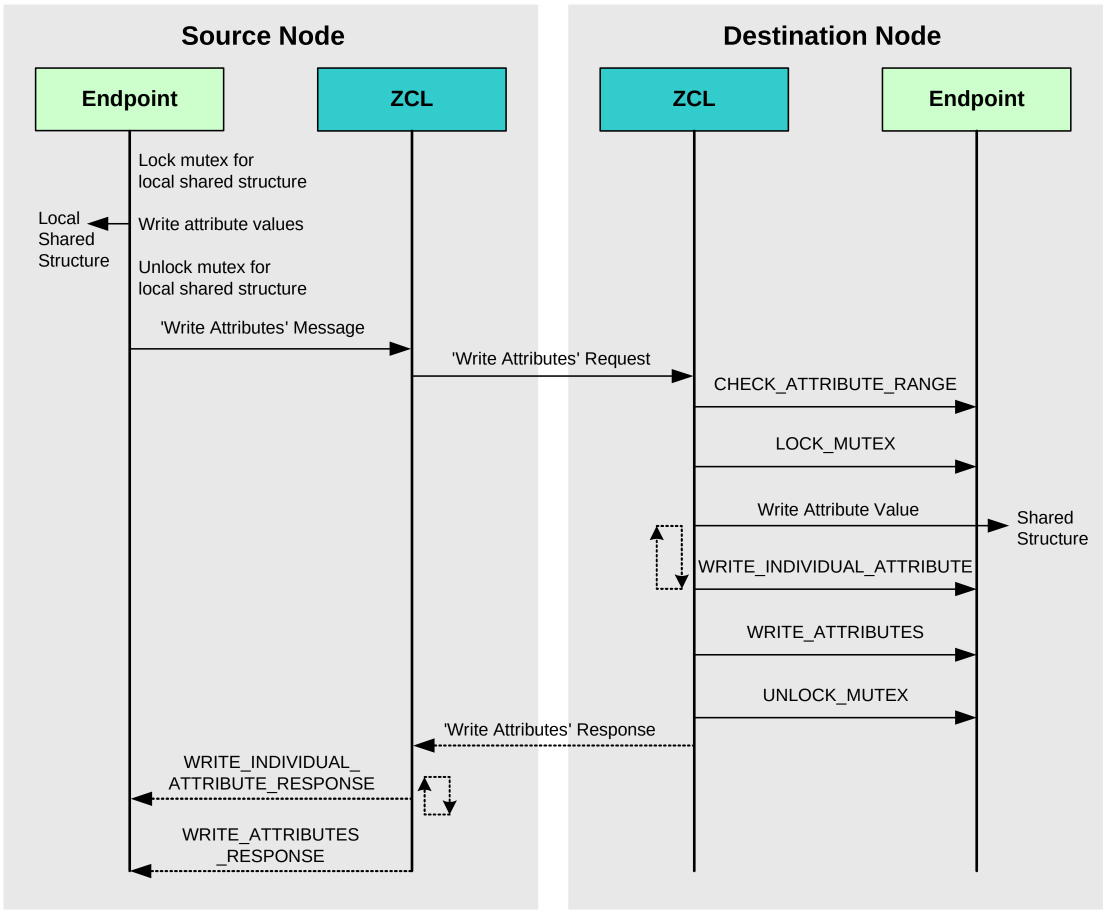

Source Node
Destination Node
Endpoint
ZCL
ZCL
Endpoint
Lock mutex for local shared structure
Local Shared Structure
Write attribute values
Unlock mutex for local shared structure
'Write Attributes' Message
'Write Attributes' Request
CHECK_ATTRIBUTE_RANGE
LOCK_MUTEX
Write Attribute Value
Shared Structure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

Figure 3: ‘Write Attributes’ Request and Response

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

4.7 Handling Stack and Timer Events

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

The events that are not cluster-specific are divided into four categories, as shown in Table 19 below - these events are described in the *ZCL User Guide (JN-UG-3077)*. Cluster-specific events are covered in the chapter for the relevant cluster.

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

Table 19: Events (Not Cluster-Specific)

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

4.8 Servicing Timing Requirements

Some clusters used by an HA application may have timing requirements which demand periodic updates. The function **eHA_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 **eHA_Update100mS()** calls the external function **vIdEffectTick()**, 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.

4.9 Time Management

A Home Automation device may need to keep track of time for its own purposes. It is not usually necessary to synchronise time between the nodes of an HA network and therefore an HA device does not normally require the Time cluster. An HA device uses 'ZCL time', which is a time in seconds maintained locally by the device.

In the NXP implementation of the ZigBee Cluster Library, ZCL time on a device is normally derived from a software timer provided by JenOS. In addition, HA requires a 100-ms timer to periodically notify the device when 100 milliseconds have passed. Typically, both of these timings are derived from the same JenOS timer. The maintenance of ZCL time and the 100-ms timing is described in the sub-sections below.

Note: The functions **vZCL_SetUTCTime()** and **vZCL_EventHandler()**, referenced below, are described in the *ZCL User Guide (JN-UG-3077)* and the function **OS_eContinueSWTimer()** is described in the *JenOS User Guide (JN-UG-3075)*.

4.9.1 Time Maintenance

ZCL time should be incremented once per second. In addition, an HA device must be prompted every 100 milliseconds to perform certain periodic operations (e.g. level transitions performed by the Level Control cluster). Normally, the same 100-ms JenOS timer is used for both timings, as follows:

1. On expiration of the 100-ms JenOS timer, an event is generated (from the hardware/software timer that drives the JenOS timer) which causes JenOS to activate an application task.
2. Within this task, the application must call **eHA_Update100mS()** on each activation of the task - this function is described in Chapter 12.
3. Every ten times that the task is activated, the application must also call **vZCL_EventHandler()** with an event type of E_ZCL_CBET_TIMER.

   This results in the timer event being passed to the ZCL once per second. On receiving each timer event, the ZCL automatically increments the ZCL time and may run cluster-specific schedulers.
4. The user task must finally resume the 100-ms timer using the JenOS function **OS_eContinueSWTimer()**.

Note 1: The function **eHA_Update100mS()** calls the external function **vIdEffectTick()**, 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.

Note 2: For more information on using the function **vZCL_EventHandler()** to pass a timer event to the ZCL, refer to the ‘Processing Events’ section of the *ZCL User Guide (JN-UG-3077)*.

4.9.2 Updating ZCL Time Following Sleep

An HA network may include nodes that conserve energy by sleeping between activities. For example:

- A switch device will normally be a sleeping End Device which wakes on an interrupt generated by the switch or dimmer hardware. If a switch supports the Identify cluster, while in identification mode it must wake once per second to generate a timer event - refer to the section on the Identify cluster in the *ZCL User Guide (JN-UG-3077)*.
- A light device is normally configured as a Router, in which case it is always active and therefore does not sleep. If a light device does sleep, it must wake at least once every 100 ms to call **eHA_Update100mS()** and also to generate a timer event once every second (see Section 4.9.1).

In the case of a device that sleeps, on waking from sleep the application should update the ZCL time using the function **vZCL_SetUTCTime()** according to the duration for which the device was asleep. This requires the sleep duration to be timed.

While sleeping, the JN5168 microcontroller normally uses its RC oscillator for timing purposes, which may not maintain the required accuracy for certain applications. In such cases, a more accurate external crystal should be used to time the sleep periods.

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

Part II:
HA Clusters

5. ZCL Clusters

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

- Basic - see Section 5.1
- Power Configuration - see Section 5.2
- Identify - see Section 5.3
- Groups - see Section 5.4
- Scenes - see Section 5.5
- On/Off - see Section 5.6
- On/Off Switch Configuration - see Section 5.7
- Level Control - see Section 5.8
- Time - see Section 5.9
- Binary Input (Basic) - see Section 5.10
- Door Lock - see Section 5.11
- Colour Control - see Section 5.12
- Illuminance Measurement - see Section 5.13
- Occupancy Sensing - see Section 5.14

The above clusters are introduced below but are fully detailed in the *ZCL User Guide (JN-UG-3077)*.

Note: In addition to the above ZCL clusters, an HA application may use the EZ-mode Commissioning module. This is also detailed in the *ZCL User Guide (JN-UG-3077).*

5.1 Basic Cluster

The Basic cluster holds basic information about a device/endpoint.

The Basic cluster has a Cluster ID of 0x0000.

It is required in HA devices as indicated in the table below.

	Server-side	Client-side
Mandatory in...	All HA devices	
Optional in...		Remote Control

Table 20: Basic Cluster in HA Devices

5.2 Power Configuration Cluster

The Power Configuration cluster provides functionality relating to the power source(s) of a device.

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

It is required in HA devices as indicated in the table below.

	Server-side	**Client-side**
Mandatory in...		
Optional in...	All HA devices	

Table 21: Power Configuration Cluster in HA Devices

5.3 Identify Cluster

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

The Identify cluster has a Cluster ID of 0x0003.

It is required in HA devices as indicated in the table below.

	Server-side	**Client-side**
Mandatory in...	All HA devices	Door Lock Controller Simple Sensor On/Off Switch On/Off Light Switch Dimmer Switch Colour Dimmer Switch Light Sensor Occupancy Sensor
Optional in...		Level Control Switch Scene Selector Remote Control

Table 22: Identify Cluster in HA Devices

5.4 Groups Cluster

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

The Groups cluster has a Cluster ID of 0x0004.

It is required in HA devices as indicated in the table below.

	Server-side	**Client-side**
Mandatory in...	On/Off Output Door Lock Simple Sensor On/Off Light Dimmable Light Colour Dimmable Light	Scene Selector Door Lock Controller
Optional in...		On/Off Switch Level Control Switch Remote Control On/Off Light Switch Dimmer Switch Colour Dimmer Switch Light Sensor Occupancy Sensor

Table 23: Groups Cluster in HA Devices

5.5 Scenes Cluster

The Scenes cluster allows values that make up a ‘scene’ to be set and retrieved.

The Scenes cluster has a Cluster ID of 0x0005.

It is required in HA devices as indicated in the table below.

	Server-side	**Client-side**
Mandatory in...	On/Off Output Door Lock Simple Sensor On/Off Light Dimmable Light Colour Dimmable Light	Scene Selector Door Lock Controller
Optional in...		On/Off Switch Level Control Switch Remote Control On/Off Light Switch Dimmer Switch Colour Dimmer Switch

Table 24: Scenes Cluster in HA Devices

5.6 On/Off Cluster

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

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

It is required in HA devices as indicated in the table below.

	Server-side	**Client-side**
Mandatory in...	On/Off Output On/Off Light Dimmable Light Colour Dimmable Light	On/Off Switch Level Control Switch On/Off Light Switch Dimmer Switch Colour Dimmer Switch
Optional in...		Remote Control

Table 25: On/Off Cluster in HA Devices

5.7 On/Off Switch Configuration Cluster

The On/Off Switch Configuration cluster allows the switch type on a device to be defined, as well as the commands to be generated when the switch is moved between its two states.

The On/Off cluster has a Cluster ID of 0x0007.

It is required in HA devices as indicated in the table below.

	Server-side	**Client-side**
Mandatory in...		
Optional in...	On/Off Switch Level Control Switch On/Off Light Switch Dimmer Switch Colour Dimmer Switch	Remote Control

Table 26: On/Off Switch Configuration Cluster in HA Devices

5.8 Level Control Cluster

The Level Control cluster is used to control the level of a physical quantity on a device (e.g. heat output).

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

It is required in HA devices as indicated in the table below.

	Server-side	Client-side
Mandatory in...	Dimmable Light Colour Dimmable Light	Level Control Switch Dimmer Switch Colour Dimmer Switch
Optional in...		Remote Control

Table 27: Level Control Cluster in HA Devices

5.9 Time Cluster

The Time cluster is used to maintain a time reference for the transactions in a ZigBee PRO network and to time-synchronise the ZigBee PRO devices.

The Time cluster has a Cluster ID of 0x000A.

It is required in HA devices as indicated in the table below.

	Server-side	Client-side
Mandatory in...		
Optional in...		All HA devices

Table 28: Time Cluster in HA Devices

5.10 Binary Input (Basic) Cluster

The Binary Input (Basic) cluster is used to read the value of a binary measurement representing the state of a two-state physical quantity.

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

It is required in HA devices as indicated in the table below.

	Server-side	Client-side
Mandatory in...	Simple Sensor	
Optional in...		

Table 29: Binary Input (Basic) Cluster in HA Devices

5.11 Door Lock Cluster

The Door Lock cluster provides an interface to a set values representing the state of a door lock and (optionally) the door.

The Door Lock cluster has a Cluster ID of 0x0101.

It is required in HA devices as indicated in the table below.

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

Table 30: Door Lock Cluster in HA Devices

In Home Automation, the Door Lock cluster is enhanced with an extra optional attribute which allows Application-level security to be used (in addition to the default Network-level security). This enhancement is described in the *ZCL User Guide (JN-UG-3077)*.

5.12 Colour Control Cluster

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

The Colour Control cluster has a Cluster ID of 0x0300.

It is required in HA devices as indicated in the table below.

	Server-side	**Client-side**
Mandatory in...	Colour Dimmable Light	Colour Dimmer Switch
Optional in...		Remote Control

Table 31: Colour Control Cluster in HA Devices

When the Colour Control cluster is used with the Scenes cluster in the HA profile, only the mandatory Colour Control cluster attributes `u16CurrentX` and `u16CurrentY` can be stored in and recalled from scenes. To enable this scenes functionality, the following definition must be added to the **zcl_options.h** file:

```
#define HA_RECALL_SCENES
```

5.13 Illuminance Measurement Cluster

The Illuminance Measurement cluster is used to interface with a set of values related to an illuminance measurement.

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

It is required in HA devices as indicated in the table below.

	Server-side	**Client-side**
Mandatory in...	Light Sensor	
Optional in...		Remote Control

Table 32: Illuminance Measurement Cluster in HA Devices

5.14 Occupancy Sensing Cluster

The Occupancy Sensing cluster is used to interface with a set of values related to occupancy sensing.

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

It is required in HA devices as indicated in the table below.

	Server-side	**Client-side**
Mandatory in...	Occupancy Sensor	
Optional in...		On/Off Light Dimmable Light Colour Dimmable Light

Table 33: Occupancy Sensing Cluster in HA Devices

6. Poll Control Cluster

This chapter describes the Poll Control cluster which is defined in the ZigBee Home Automation profile, and provides an interface for remotely controlling the rate at which a ZigBee End Device polls its parent for data.

The Poll Control cluster has a Cluster ID of 0x0020.

6.1 Overview

An End Device cannot receive data packets directly, as it may be asleep when a packet arrives. The data packets for an End Device are therefore buffered by the device's parent and the End Device polls its parent for data while awake. An individual data packet will only be held on the parent node for a maximum of 7.68 seconds and if many packets for the End Device are expected over a short period of time, the End Device should retrieve these packets as quickly as possible. An End Device can implement two polling modes, which are dependent on the poll interval (time-period between consecutive polls):

- **Normal poll mode:** A long poll interval is used - this mode is appropriate when the End Device is not expecting data packets
- **Fast poll mode:** A short poll interval is used - this mode is appropriate when the End Device is expecting data packets

The End Device may enable fast poll mode itself when it is expecting data packets (e.g. after it has requested data from remote nodes). The Poll Control cluster allows fast poll mode to be selected from a remote control device to force the End Device to be more receptive to data packets (e.g. when a download to the End Device involving a large number of unsolicited data packets is to be initiated).

The two sides of the cluster are located as follows:

- The cluster server is implemented on the End Device to be controlled
- The cluster client is implemented on the remote controller device

The cluster server (End Device) periodically checks whether the cluster client (remote controller) requires the poll mode to be changed. This 'check-in' method is used since an unsolicited instruction from the controller may arrive when the End Device is asleep. The automatic 'check-ins' are conducted with all the remote endpoints (on controller nodes) to which the local endpoint (on which the cluster resides) is bound.

The cluster is enabled by defining CLD_POLL_CONTROL in the **zcl_options.h** file - see Section 3.5.1. Further compile-time options for the Poll Control cluster are detailed in Section 6.10.

6.2 Cluster Structure and Attributes

The Poll Control cluster server attributes are contained in the following `tsCLD_PollControl` structure:

```
typedef struct
{

    zuint32     u32CheckinInterval;
    zuint32     u32LongPollInterval;
    zuint16     u16ShortPollInterval;
    zuint16     u16FastPollTimeout;

#ifdef CLD_POLL_CONTROL_ATTR_CHECKIN_INTERVAL_MIN
    zuint32     u32CheckinIntervalMin;
#endif

#ifdef CLD_POLL_CONTROL_ATTR_LONG_POLL_INTERVAL_MIN
    zuint32     u32LongPollIntervalMin;
#endif

#ifdef CLD_POLL_CONTROL_ATTR_FAST_POLL_TIMEOUT_MAX
    zuint16     u16FastPollTimeoutMax;
#endif

} tsCLD_PollControl;
```

where:

- `u32CheckinInterval` is the ‘check-in interval’, used by the server in checking whether a client requires the poll mode to be changed - this is the period, in quarter-seconds, between consecutive checks. The valid range of values is 1 to 7208960. A user-defined minimum value for this attribute can be set via the optional attribute `u32CheckinIntervalMin` (see below). Zero is a special value indicating that the Poll Control cluster server is disabled. The default value is 14400 (1 hour).
- `u32LongPollInterval` is the ‘long poll interval’ of the End Device, employed when operating in normal poll mode - this is the period, in quarter-seconds, between consecutive polls of the parent for data. The valid range of values is 4 to 7208960. A user-defined minimum value for this attribute can be set via the optional attribute `u32LongPollIntervalMin` (see below). 0xFFFF is a special value indicating that the long poll interval is unknown/undefined. The default value is 20 (5 seconds).
- `u16ShortPollInterval` is the ‘short poll interval’ of the End Device, employed when operating in fast poll mode - this is the period, in quarter-seconds, between consecutive polls of the parent for data. The valid range of values is 1 to 65535 and the default value is 2 (0.5 seconds).

- `u16FastPollTimeout` is the 'fast poll timeout' representing the time-interval, in quarter-seconds, for which the server should normally stay in fast poll mode (unless over-ridden by a client command). The valid range of values is 1 to 65535. It is recommended that this timeout is greater than 7.68 seconds. A user-defined maximum value for this attribute can be set via the optional attribute `u16FastPollTimeoutMax` (see below). The default value is 40 (10 seconds).
- `u32CheckinIntervalMin` is an optional lower limit on the 'check-in interval' defined by `u32CheckinInterval`. This limit can be used to ensure that the interval is not inadvertently set to a low value which will quickly drain the energy resources of the End Device node.
- `u32LongPollIntervalMin` is an optional lower limit on the 'long poll interval' defined by `u32LongPollInterval`. This limit can be used to ensure that the interval is not inadvertently set (e.g. by another device) to a low value which will quickly drain the energy resources of the End Device node.
- `u16FastPollTimeoutMax` is an optional upper limit on the 'fast poll timeout' defined by `u16FastPollTimeout`. This limit can be used to ensure that the interval is not inadvertently set (e.g. by another device) to a high value which will quickly drain the energy resources of the End Device node.

Note 1: Valid ranges (maximum and minimum values) for the four mandatory attributes can alternatively be set using macros in the **zcl_options.h** file, as described in Section 6.10. Some of these macros can only be used when the equivalent optional attribute is disabled.

Note 2: For general guidance on attribute settings, refer to Section 6.3. Configuration through the attributes is also described in Section 6.4.2.

6.3 Attribute Settings

In assigning user-defined values to the mandatory attributes, the following inequality should be obeyed:

`u32CheckinInterval` ≥ `u32LongPollInterval` ≥ `u16ShortPollInterval`

In addition, the mandatory attribute `u16FastPollTimeout` should not be set to an excessive value for self-powered nodes, as fast poll mode can rapidly drain the stored energy of a node (e.g. the battery).

The three optional attributes can be used to ensure that the values of the corresponding mandatory attributes are kept within reasonable limits, to prevent the rapid depletion of the energy resources of the node. If required, the optional attributes must be enabled and initialised in the compile-time options (see Section 6.10).

Minimum and maximum values for all the mandatory attributes can alternatively be set using the compile-time options (again, refer to Section 6.10).

6.4 Poll Control Operations

This section describes the main operations to be performed on the Poll Control cluster server (End Device) and client (controller).

6.4.1 Initialisation

The Poll Control cluster must be initialised on both the cluster server and client. This can be done using the function **eCLD_PollControlCreatePollControl()**, which creates an instance of the Poll Control cluster on a local endpoint.

If you are using a standard ZigBee device which includes the Poll Control cluster, the above function will be automatically called by the initialisation function for the device. You only need to call **eCLD_PollControlCreatePollControl()** explicitly when setting up a custom endpoint containing one or more selected clusters (rather than the whole set of clusters supported by a standard ZigBee device).

6.4.2 Configuration

When initialised, the Poll Control cluster will adopt the attribute values that are pre-set in the `tsCLD_PollControl` structure (see Section 6.2). For the optional attributes, values can be set in the file **zcl_options.h** (see Section 6.10).

The mandatory attributes (and related optional attributes) are as follows:

- **Long Poll Interval (`u32LongPollInterval`)**: This is the polling period used in normal poll mode, expressed in quarter-seconds, with a default value of 20 (5 seconds). The attribute has a valid range of 4 to 7208960 but a user-defined minimum value for this attribute can be set via the optional 'long poll interval maximum' attribute (`u32LongPollIntervalMin`). This limit can be used to ensure that the interval is not inadvertently set (e.g. by another device) to a low value which will quickly drain the energy resources of the End Device node. Alternatively, minimum and maximum values can be specified through the compile-time options (see Section 6.10).
- **Short Poll Interval (`u16ShortPollInterval`)**: This is the polling period used in fast poll mode, expressed in quarter-seconds, with a default value of 2 (0.5 seconds). The attribute has a valid range of 1 to 65535. User-defined minimum and maximum values for this attribute can be specified through the compile-time options (see Section 6.10).
- **Fast Poll Timeout (`u16FastPollTimeout`)**: This is the time-interval for which the server should normally stay in fast poll mode (unless over-ridden by a client command), expressed in quarter-seconds, with a default value of 40 (10 seconds). It is recommended that this timeout is greater than 7.68 seconds. The valid range of values is 1 to 65535 but a user-defined maximum value for this attribute can be set via the optional 'fast poll timeout maximum' attribute (`u16FastPollTimeoutMax`). This limit can be used to ensure that the interval is not inadvertently set (e.g. by another device) to a high value which will quickly drain the energy resources of the End Device node. Alternatively, minimum and maximum values can be specified through the compile-time options (see Section 6.10).

- **Check-in Interval (`u32CheckinInterval`)**: This is the period between the server's checks of whether a client requires the poll mode to be changed, expressed in quarter-seconds, with a default value of 14400 (1 hour). It should be greater than the 'long poll interval' (see above). Zero is a special value indicating that the Poll Control cluster server is disabled. Otherwise, the valid range of values is 1 to 7208960 but a user-defined minimum value for this attribute can be set via the optional 'check-in interval minimum' attribute (`u32CheckinIntervalMin`). This limit can be used to ensure that the interval is not inadvertently set to a low value which will quickly drain the energy resources of the End Device node. Alternatively, minimum and maximum values can be specified through the compile-time options (see Section 6.10).

The Poll Control cluster server can also be configured by the server application at run-time by writing to the relevant attribute(s) using the **eCLD_PollControlSetAttribute()** function (which must be called separately for each attribute to be modified). If used, this function must be called after the cluster has been initialised (see Section 6.4.1).

Changes to certain attributes can also be initiated remotely from the cluster client (controller) using the following functions:

- **eCLD_PollControlSetLongPollIntervalSend()**: The client application can use this function to submit a request to set the 'long poll interval' attribute on the server to a specified value. This function causes a 'Set Long Poll Interval' command to be sent to the relevant End Device. If the new value is acceptable, the cluster server will automatically update the attribute.
- **eCLD_PollControlSetShortPollIntervalSend()**: The client application can use this function to submit a request to set the 'short poll interval' attribute on the server to a specified value. This function causes a 'Set Short Poll Interval' command to be sent to the relevant End Device. If the new value is acceptable, the cluster server will automatically update the attribute.

In both of the above cases, a response will only be sent back to the client if the new value is not acceptable, in which case a ZCL 'default response' will be sent indicating an invalid value.

Use of the above two functions requires the corresponding commands to be enabled in the compile-time options, as described in Section 6.10.

Note: Changes to attribute values initiated by either the server application or client application will take effect immediately. So, for example, if the End Device is operating in fast poll mode when the 'short poll interval' is modified, the polling period will be immediately re-timed to the new value. If the modified attribute is not related to the currently operating poll mode, the change will be implemented the next time the relevant poll mode is started.

Before the first scheduled 'check-in' (after one hour, by default), the End Device application should set up bindings between the local endpoint on which the cluster resides and the relevant endpoint on each remote controller node with which the End Device will operate. These bindings will be used in sending the 'Check-in' commands.

6.4.3 Operation

After initialisation, the Poll Control cluster server on the End Device will begin to operate in normal poll mode and will need to perform the following activities (while the End Device is awake):

- Periodically poll the parent for data packets at a rate determined by the ‘long poll interval’
- Periodically check whether any bound cluster clients require the server to enter fast poll mode, with ‘check-ins’ at a rate determined by the ‘check-in interval’

The server application must provide the cluster with timing prompts for the above periodic activities. These prompts are produced by periodically calling the function **eCLD_PollControlUpdate()**. Since the periods of the above activities are defined in terms of quarter-seconds, this function must be called every quarter-second and the application must provide a 250-ms software timer to schedule these calls. Any poll or check-in that is due when this function is called will be automatically performed by the cluster server.

The End Device will operate in normal poll mode until either it puts itself into fast poll mode (e.g. when it is expecting responses to a request) or the controller (client) requests the End Device to enter fast poll mode (e.g. when a data download to the End Device is going to be performed). As indicated above, such a request from the client is raised as the result of the server performing periodic ‘check-ins’ with the client.

On receiving a ‘check-in’ command, an E_CLD_POLL_CONTROL_CMD_CHECK_IN event is generated on the client. The client application must then fill in the `tsCLD_PollControl_CheckinResponsePayload` structure (see Section 6.9.2) of the event, indicating whether fast poll mode is required. A response will then be automatically sent back to the server.

After sending the initial Check-in command, the server will wait for up to 7.68 seconds for a response (if no response is received in this time, the server is free to continue in normal poll mode). If a response is received, the event E_CLD_POLL_CONTROL_CMD_CHECK_IN will be generated on the server, which will automatically put the End Device into fast poll mode. The response payload (see Section 6.9.2) contains an optional timeout value which, if used, specifies the length of time that the device should remain in fast poll mode (this timeout value will be used instead of the one specified through the ‘fast poll timeout’ attribute). If the response payload specifies an out-of-range timeout value, the server will send a ZCL default response with status INVALID_VALUE to the client. In the case of multiple clients (controllers) that have specified different timeout values, the server will use the largest timeout value received.

When the End Device is in fast poll mode, the client application can request the cluster server to exit fast poll mode immediately (before the timeout expires) by calling the function **eCLD_PollControlFastPollStopSend()**.

6.5 Poll Control Events

The Poll Control cluster has its own events that are handled through the callback mechanism outlined in Section 4.7 (and fully detailed in the *ZCL User Guide (JN-UG-3077)*). The cluster contains its own event handler. However, if a device uses this cluster then application-specific Poll Control event handling must be included in the user-defined callback function for the associated endpoint, where this callback function is registered through the relevant endpoint registration function. This callback function will then be invoked when a Poll Control event occurs and needs the attention of the application.

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

```
typedef struct
{
    uint8        u8CommandId;
    union
    {
        tsCLD_PollControl_CheckinResponsePayload *psCheckinResponsePayload;
    #ifdef CLD_POLL_CONTROL_CMD_SET_LONG_POLL_INTERVAL
        tsCLD_PollControl_SetLongPollIntervalPayload
                                                 *psSetLongPollIntervalPayload;
    #endif
    #ifdef CLD_POLL_CONTROL_CMD_SET_SHORT_POLL_INTERVAL
        tsCLD_PollControl_SetShortPollIntervalPayload
                                                 *psSetShortPollIntervalPayload;
    #endif
    } uMessage;
} tsCLD_PollControlCallBackMessage;
```

The above structure is fully described in Section 6.9.1.

When a Poll Control event occurs, one of the command types listed in Table 34 is specified through the `u8CommandId` field of the structure `tsCLD_PollControlCallBackMessage`. This command type determines which command payload is used from the union `uMessage`.

u8CommandId Enumeration	Description/Payload Type
On Client	
E_CLD_POLL_CONTROL_CMD_CHECK_IN	A Check-in command has been received by the client.
On Server	
E_CLD_POLL_CONTROL_CMD_CHECK_IN	A Check-in Response has been received by the server, following a previously sent Check-In command. **tsCLD_PollControl_CheckinResponsePayload**
E_CLD_POLL_CONTROL_CMD_FAST_POLL_STOP	A 'Fast Poll Stop' command has been received by the server.
E_CLD_POLL_CONTROL_CMD_SET_LONG_POLL_INTERVAL	A 'Set Long Poll Interval' command has been received by the server. **tsCLD_PollControl_SetLongPollIntervalPayload**
E_CLD_POLL_CONTROL_CMD_SET_SHORT_POLL_INTERVAL	A 'Set Short Poll Interval' command has been received by the server. **tsCLD_PollControl_SetShortPollIntervalPayload**

Table 34: Poll Control Command Types (Events)

6.6 Functions

The Poll Control cluster functions provided in the HA API are described in the following three sub-sections, according to the side(s) of the cluster on which they can be used:

- Server/client function are described in Section 6.6.1
- Server functions are described in Section 6.6.2
- Client functions are described in Section 6.6.3

6.6.1 Server/Client Function

The following Poll Control cluster function is provided in the HA API and can be used on either a cluster server or cluster client:

Function	Page
eCLD_PollControlCreatePollControl	83

eCLD_PollControlCreatePollControl

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

Description

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

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

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

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

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

The array declaration should be as follows:

```
uint8 au8PollControlClusterAttributeControlBits[
CLD_POLL_CONTROL_MAX_NUMBER_OF_ATTRIBUTE];
```

The function will initialise the array elements to zero.

Parameters

psClusterInstance	Pointer to structure containing information about the cluster instance to be created (see the *ZCL User Guide (JN-UG-3077)*). This structure will be updated by the function by initialising individual structure fields.
bIsServer	Type of cluster instance (server or client) to be created: TRUE - server FALSE - client
psClusterDefinition	Pointer to structure indicating the type of cluster to be created (see the *ZCL User Guide (JN-UG-3077)*). In this case, this structure must contain the details of the Poll Control cluster. This parameter can refer to a pre-filled structure called `sCLD_PollControl` which is provided in the **PollControl.h** file.
pvEndPointSharedStructPtr	Pointer to the shared structure used for attribute storage. This parameter should be the address of the structure of type `tsCLD_PollControl` which defines the attributes of the Poll Control cluster. The function will initialise the attributes with default values.
pu8AttributeControlBits	Pointer to an array of **uint8** values, with one element for each attribute in the cluster (see above).
psCustomDataStructure	Pointer to a structure containing the storage for internal functions of the cluster (see Section 6.9.5)

Returns

E_ZCL_SUCCESS
E_ZCL_FAIL
E_ZCL_ERR_PARAMETER_NULL
E_ZCL_ERR_INVALID_VALUE

6.6.2 Server Functions

The following Poll Control cluster functions are provided in the HA API and can be used on a cluster server only:

Function	Page
eCLD_PollControlUpdate	86
eCLD_PollControlSetAttribute	87

eCLD_PollControlUpdate

```
teZCL_Status eCLD_PollControlUpdate(void);
```

Description

This function can be used on a cluster server to update the timing status for the following periodic activities:

- polling of the parent for a data packet
- 'check-ins' with the client to check for a required change in the poll mode

The function should be called once per quarter-second and the application should provide a 250-ms timer to prompt these function calls.

Any poll or check-in that is due when this function is called will be automatically performed by the cluster server.

Parameters

None

Returns

E_ZCL_SUCCESS
E_ZCL_FAIL

eCLD_PollControlSetAttribute

```
teZCL_Status eCLD_PollControlSetAttribute(
                                    uint8 u8SourceEndPointId,
                                    uint8 u8AttributeId,
                                    uint32 u32AttributeValue);
```

Description

This function can be used on a cluster server to write to an attribute of the Poll Control cluster. The function will write to the relevant field of the `tsCLD_PollControl` structure (detailed in Section 6.2). The attribute to be accessed is specified using its attribute identifier - enumerations are provided (see Section 6.8.1).

Therefore, this function can be used to change the configuration of the Poll Control cluster. The change will take effect immediately. So, for example, if the End Device is in normal poll mode when the 'long poll interval' is modified, the polling period will be immediately re-timed to the new value. If the modified attribute is not related to the currently operating poll mode, the change will be implemented the next time the relevant poll mode is started.

The specified value of the attribute is validated by the function. If this value is out-of-range for the attribute, the status E_ZCL_ERR_INVALID_VALUE is returned.

Parameters

u8SourceEndPointId	Number of local endpoint on which cluster resides
u8AttributeId	Identifier of attribute to be written to (see Section 6.8.1)
u32AttributeValue	Value to be written to attribute

Returns

E_ZCL_SUCCESS

E_ZCL_FAIL

E_ZCL_ERR_INVALID_VALUE

E_ZCL_DENY_ATTRIBUTE_ACCESS

6.6.3 Client Functions

The following Poll Control cluster functions are provided in the HA API and can be used on a cluster client only:

Function	Page
eCLD_PollControlSetLongPollIntervalSend	89
eCLD_PollControlSetShortPollIntervalSend	91
eCLD_PollControlFastPollStopSend	93

eCLD_PollControlSetLongPollIntervalSend

```
teZCL_Status eCLD_PollControlSetLongPollIntervalSend(
          uint8 u8SourceEndPointId,
          uint8 u8DestinationEndPointId,
          tsZCL_Address *psDestinationAddress,
          uint8 *pu8TransactionSequenceNumber,
          tsCLD_PollControl_SetLongPollIntervalPayload
                                                  *psPayload);
```

Description

This function can be used on a cluster client to send a ‘Set Long Poll Interval’ command to the cluster server. This command requests the ‘long poll interval’ for normal poll mode on the End Device to be set to the specified value.

On receiving the command, the ‘long poll interval’ attribute is only modified by the server if the specified value is within the valid range for the attribute (including greater than or equal to the optional user-defined minimum, if set) - see Section 6.2. If this is not the case, the server replies to the client with a ZCL ‘default response’ indicating an invalid value.

The change will take effect immediately. So, if the End Device is in normal poll mode when the ‘long poll interval’ is modified, the polling period will be immediately re-timed to the new value.

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

Parameters

u8SourceEndPointId	Number of local endpoint on which cluster client resides
u8DestinationEndPointId	Number of remote endpoint on which cluster server resides
psDestinationAddress	Pointer to a structure containing the destination address of the server node
pu8TransactionSequenceNumber	Pointer to a location to receive the Transaction Sequence Number (TSN) of the message
psPayload	Pointer to structure containing the payload for the command (see Section 6.9.3), including the desired long poll interval

Returns

E_ZCL_SUCCESS

E_ZCL_FAIL

E_ZCL_ERR_PARAMETER_NULL

E_ZCL_ERR_EP_RANGE

E_ZCL_ERR_EP_UNKNOWN

E_ZCL_ERR_CLUSTER_NOT_FOUND

E_ZCL_ERR_ZBUFFER_FAIL

E_ZCL_ERR_ZTRANSMIT_FAIL

eCLD_PollControlSetShortPollIntervalSend

```
teZCL_Status eCLD_PollControlSetShortPollIntervalSend(
        uint8 u8SourceEndPointId,
        uint8 u8DestinationEndPointId,
        tsZCL_Address *psDestinationAddress,
        uint8 *pu8TransactionSequenceNumber,
        tsCLD_PollControl_SetShortPollIntervalPayload
                                        *psPayload);
```

Description

This function can be used on a cluster client to send a ‘Set Short Poll Interval’ command to the cluster server. This command requests the ‘short poll interval’ for fast poll mode on the End Device to be set to the specified value.

On receiving the command, the ‘short poll interval’ attribute is only modified by the server if the specified value is within the valid range for the attribute (including greater than or equal to the optional user-defined minimum, if set) - see Section 6.2. If this is not the case, the server replies to the client with a ZCL ‘default response’ indicating an invalid value.

The change will take effect immediately. So, if the End Device is in fast poll mode when the ‘short poll interval’ is modified, the polling period will be immediately re-timed to the new value.

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

Parameters

u8SourceEndPointId	Number of local endpoint on which cluster client resides
u8DestinationEndPointId	Number of remote endpoint on which cluster server resides
psDestinationAddress	Pointer to a structure containing the destination address of the server node
pu8TransactionSequenceNumber	Pointer to a location to receive the Transaction Sequence Number (TSN) of the message
psPayload	Pointer to structure containing the payload for the command (see Section 6.9.4), including the desired short poll interval

Returns

E_ZCL_SUCCESS
E_ZCL_FAIL
E_ZCL_ERR_PARAMETER_NULL
E_ZCL_ERR_EP_RANGE
E_ZCL_ERR_EP_UNKNOWN
E_ZCL_ERR_CLUSTER_NOT_FOUND
E_ZCL_ERR_ZBUFFER_FAIL
E_ZCL_ERR_ZTRANSMIT_FAIL

eCLD_PollControlFastPollStopSend

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

Description

This function can be used on a cluster client to send a ‘Fast Poll Stop’ command to the cluster server. This command is intended to abort a fast poll mode episode which has been started on the server as the result of a ‘Check-in Response’. Therefore, the command allows fast poll mode to be exited before the mode’s timeout is reached.

The cluster server will only stop fast poll mode on the destination End Device if a matching ‘Fast Poll Stop’ command has been received for every request to start the current episode of fast poll mode. Therefore, if the current fast poll mode episode resulted from multiple start requests from multiple clients, the episode cannot be prematurely stopped (before the timeout is reached) unless a ‘Fast Poll Stop’ command is received from each of those clients.

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

Parameters

u8SourceEndPointId	Number of local endpoint on which cluster client resides
u8DestinationEndPointId	Number of remote endpoint on which cluster server resides
psDestinationAddress	Pointer to a structure containing the destination address of the server node
pu8TransactionSequenceNumber	Pointer to a location to receive the Transaction Sequence Number (TSN) of the message

Returns

E_ZCL_SUCCESS

E_ZCL_FAIL

E_ZCL_ERR_PARAMETER_NULL

E_ZCL_ERR_EP_RANGE

E_ZCL_ERR_EP_UNKNOWN

E_ZCL_ERR_CLUSTER_NOT_FOUND

E_ZCL_ERR_ZBUFFER_FAIL

E_ZCL_ERR_ZTRANSMIT_FAIL

6.7 Return Codes

The Poll Control cluster functions use the ZCL return codes defined in the *ZCL User Guide (JN-UG-3077)*.

6.8 Enumerations

6.8.1 ‘Attribute ID’ Enumerations

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

```
typedef enum PACK
{
    E_CLD_POLL_CONTROL_ATTR_ID_CHECKIN_INTERVAL = 0x0000,
    E_CLD_POLL_CONTROL_ATTR_ID_LONG_POLL_INTERVAL,
    E_CLD_POLL_CONTROL_ATTR_ID_SHORT_POLL_INTERVAL,
    E_CLD_POLL_CONTROL_ATTR_ID_FAST_POLL_TIMEOUT,
    E_CLD_POLL_CONTROL_ATTR_ID_CHECKIN_INTERVAL_MIN,
    E_CLD_POLL_CONTROL_ATTR_ID_LONG_POLL_INTERVAL_MIN,
    E_CLD_POLL_CONTROL_ATTR_ID_FAST_POLL_TIMEOUT_MAX
} teCLD_PollControl_Cluster_AttrID;
```

6.8.2 ‘Command’ Enumerations

The following enumerations represent the commands that can be generated by the Poll Control cluster.

```
typedef enum PACK
{
    E_CLD_POLL_CONTROL_CMD_CHECK_IN = 0x00,
    E_CLD_POLL_CONTROL_CMD_FAST_POLL_STOP,
    E_CLD_POLL_CONTROL_CMD_SET_LONG_POLL_INTERVAL,
    E_CLD_POLL_CONTROL_CMD_SET_SHORT_POLL_INTERVAL,
} teCLD_PollControl_CommandID;
```

The above enumerations are used to indicate types of Poll Control cluster events and are described in Section 6.5.

6.9 Structures

6.9.1 tsCLD_PPCallBackMessage

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

```
typedef struct
{
    uint8          u8CommandId;
    union
    {
        tsCLD_PollControl_CheckinResponsePayload *psCheckinResponsePayload;
    #ifdef CLD_POLL_CONTROL_CMD_SET_LONG_POLL_INTERVAL
        tsCLD_PollControl_SetLongPollIntervalPayload
                                                 *psSetLongPollIntervalPayload;
    #endif
    #ifdef CLD_POLL_CONTROL_CMD_SET_SHORT_POLL_INTERVAL
        tsCLD_PollControl_SetShortPollIntervalPayload
                                                 *psSetShortPollIntervalPayload;
    #endif
    } uMessage;
} tsCLD_PollControlCallBackMessage;
```

where:

- `u8CommandId` indicates the type of Poll Control command that has been received, one of:
  - E_CLD_POLL_CONTROL_CMD_CHECK_IN
  - E_CLD_POLL_CONTROL_CMD_FAST_POLL_STOP
  - E_CLD_POLL_CONTROL_CMD_SET_LONG_POLL_INTERVAL
  - E_CLD_POLL_CONTROL_CMD_SET_SHORT_POLL_INTERVAL

  If they are required, the last two commands must be enabled in the compile-time options, as described in Section 6.10.
- `uMessage` is a union containing the command payload, as one of (depending on the value of `u8CommandId`):
  - `psCheckinResponsePayload` is a pointer to the payload of a 'Check-in Response' (see Section 6.9.2)
  - `psSetLongPollIntervalPayload` is a pointer to the payload of a 'Set Long Poll Interval' command (see Section 6.9.3)
  - `psSetShortPollIntervalPayload` is a pointer to the payload of a 'Set Short Poll Interval' command (see Section 6.9.4)

The command payload for each command type is indicated in Table 34 in Section 6.5.

6.9.2 tsCLD_PollControl_CheckinResponsePayload

This structure contains the payload of a ‘Check-in Response’, which is sent from the client to the server in reply to a ‘Check-in’ command from the server.

```
typedef struct
{
    zbool       bStartFastPolling;
    zuint16     u16FastPollTimeout;
}tsCLD_PollControl_CheckinResponsePayload;
```

where:

- `bStartFastPolling` is a boolean indicating whether or not the End Device is required to enter fast poll mode:
  - TRUE: Enter fast poll mode
  - FALSE: Continue in normal poll mode
- `u16FastPollTimeout` is an optional fast poll mode timeout, in quarter-seconds, in the range 1 to 65535 - that is, the period of time for which the End Device should remain in fast poll mode (if this mode is requested through `bStartFastPolling`). Zero is a special value which indicates that the value of the ‘fast poll timeout’ attribute should be used instead (see Section 6.2). If a non-zero value is specified then this value will over-ride the ‘fast poll timeout’ attribute (but will not over-write it).

6.9.3 tsCLD_PollControl_SetLongPollIntervalPayload

This structure contains the payload of a ‘Set Long Poll Interval’ command, which is sent from the client to the server to request a new ‘long poll interval’ for use in normal poll mode.

```
typedef struct
{
     zuint32     u32NewLongPollInterval;
}tsCLD_PollControl_SetLongPollIntervalPayload;
```

where `u32NewLongPollInterval` is the required value of the ‘long poll interval’, in quarter-seconds, in the range 4 to 7208960. This value will be used to over-write the corresponding cluster attribute if the specified value is within the valid range for the attribute (including greater than or equal to the optional user-defined minimum, if set).

To use the ‘Set Long Poll Interval’ command, it must be enabled in the compile-time options, as described in Section 6.10.

6.9.4 tsCLD_PollControl_SetShortPollIntervalPayload

This structure contains the payload of a ‘Set Short Poll Interval’ command, which is sent from the client to the server to request a new ‘short poll interval’ for use in fast poll mode.

```
typedef struct
{
     zuint16      u16NewShortPollInterval;
}tsCLD_PollControl_SetShortPollIntervalPayload;
```

where `u16NewShortPollInterval` is the required value of the ‘short poll interval’, in quarter-seconds, in the range 1 to 65535. This value will be used to over-write the corresponding cluster attribute if the specified value is within the valid range for the attribute (including greater than or equal to the optional user-defined minimum, if set).

To use the ‘Set Short Poll Interval’ command, it must be enabled in the compile-time options, as described in Section 6.10.

6.9.5 tsCLD_PollControlCustomDataStructure

The Poll Control cluster requires extra storage space to be allocated to be used by internal functions. The structure definition for this storage is shown below:

```
typedef struct
{
#ifdef POLL_CONTROL_SERVER
    tsCLD_PollControlParameters         sControlParameters;
#endif
    tsZCL_ReceiveEventAddress           sReceiveEventAddress;
    tsZCL_CallBackEvent                 sCustomCallBackEvent;
    tsCLD_PollControlCallBackMessage    sCallBackMessage;
} tsCLD_PollControlCustomDataStructure;
```

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

6.10 Compile-Time Options

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

To enable the Poll Control cluster in the code to be built, it is necessary to add the following line to the file:

```
#define CLD_POLL_CONTROL
```

In addition, to enable the cluster as a client or server, it is also necessary to add one of the following lines to the same file:

```
#define POLL_CONTROL_SERVER
#define POLL_CONTROL_CLIENT
```

The following options can also be configured at compile-time in the **zcl_options.h** file.

Enable and Set Optional Server Attributes

To enable and assign a value (`t` quarter-seconds) to the optional Check-in Interval Minimum (`u32CheckinIntervalMin`) attribute, add the line:

```
#define CLD_POLL_CONTROL_ATTR_CHECKIN_INTERVAL_MIN t
```

To enable and assign a value (`t` quarter-seconds) to the optional Long Poll Interval Minimum (`u32LongPollIntervalMin`) attribute, add the line:

```
#define CLD_POLL_CONTROL_ATTR_LONG_POLL_INTERVAL_MIN t
```

To enable and assign a value (`t` quarter-seconds) to the optional Fast Poll Timeout Maximum (`u16FastPollTimeoutMax`) attribute, add the line:

```
#define CLD_POLL_CONTROL_ATTR_FAST_POLL_TIMEOUT_MAX t
```

Note: For further information on the above optional server attributes, refer to Section 6.2.

Set Valid Range for ‘Check-in Interval’

To set the maximum possible ‘check-in interval’ (to `t` quarter-seconds), add the line:

```
#define CLD_POLL_CONTROL_CHECKIN_INTERVAL_MAX t
```

The default value is 7208960.

To set the minimum possible ‘check-in interval’ (to `t` quarter-seconds), add the line:

```
#define CLD_POLL_CONTROL_CHECKIN_INTERVAL_MIN t
```

The default value is 0.

This minimum value is only applied if the Check-in Interval Minimum attribute (`u32CheckinIntervalMin`) is not enabled.

Set Valid Range for ‘Fast Poll Timeout’

To set the maximum possible ‘fast poll timeout’ (to `t` quarter-seconds), add the line:

```
#define CLD_POLL_CONTROL_FAST_POLL_TIMEOUT_MAX t
```

The default value is 65535.

To set the minimum possible ‘fast poll timeout’ (to `t` quarter-seconds), add the line:

```
#define CLD_POLL_CONTROL_FAST_POLL_TIMEOUT_MIN t
```

The default value is 1.

This maximum value is only applied if the Fast Poll Timeout Maximum attribute (`u16FastPollTimeoutMax`) is not enabled.

Set Valid Range for ‘Long Poll Interval’

To set the maximum possible ‘long poll interval’ (to `t` quarter-seconds), add the line:

```
#define CLD_POLL_CONTROL_LONG_POLL_INTERVAL_MAX t
```

The default value is 7208960.

To set the minimum possible ‘long poll interval’ (to `t` quarter-seconds), add the line:

```
#define CLD_POLL_CONTROL_LONG_POLL_INTERVAL_MIN t
```

The default value is 4.

This minimum value is only applied if the Long Poll Interval Minimum attribute (`u32LongPollIntervalMin`) is not enabled.

Set Valid Range for ‘Short Poll Interval’

To set the maximum possible ‘short poll interval’ (to `t` quarter-seconds), add the line:

```
#define CLD_POLL_CONTROL_SHORT_POLL_INTERVAL_MAX t
```

The default value is 65535.

To set the minimum possible ‘short poll interval’ (to `t` quarter-seconds), add the line:

```
#define CLD_POLL_CONTROL_SHORT_POLL_INTERVAL_MIN t
```

The default value is 1.

Enable Optional Commands

To enable the optional ‘Set Long Poll Interval’ command, add the line:

```
#define CLD_POLL_CONTROL_CMD_SET_LONG_POLL_INTERVAL
```

To enable the optional ‘Set Short Poll Interval’ command, add the line:

```
#define CLD_POLL_CONTROL_CMD_SET_SHORT_POLL_INTERVAL
```

Maximum Number of Clients

To set the maximum number of clients for a server to n, add the line:

```
#define CLD_POLL_CONTROL_NUMBER_OF_MULTIPLE_CLIENTS n
```

This is the maximum number of clients from which the server can handle Check-in Responses. It should be equal to the capacity (number of entries) of the binding table created on the server device to accommodate bindings to client devices (where this size is set in a ZigBee network parameter using the ZPS Configuration Editor).

7. Power Profile Cluster

This chapter describes the Power Profile cluster which is defined in the ZigBee Home Automation profile, and provides an interface between a home appliance (e.g. a washing machine) and the controller of an energy management system.

The Power Profile cluster has a Cluster ID of 0x001A.

7.1 Overview

The Power Profile cluster allows an appliance, the cluster server, to provide its expected power usage data to a controller, the cluster client. This 'power profile' represents the predicted 'energy footprint' of the appliance, and may be used by the controller to schedule and control the operation of the appliance. It may be requested by the client or provided unsolicited by the server.

The cluster is enabled by defining CLD_PP in the **zcl_options.h** file - see Section 3.5.1. Further compile-time options for the Power Profile cluster are detailed in Section 7.10.

Note: The Power Profile cluster requires the Appliance Control cluster for the implementation of status notifications and power management commands. The Appliance Control cluster is described in Chapter 8.

7.2 Cluster Structure and Attributes

The Power Profile cluster server is contained in the following `tsCLD_PP` structure:

```
typedef struct
{
    zuint8   u8TotalProfileNum;
    zbool    bMultipleScheduling;
    zbmap8   u8EnergyFormatting;
    zbool    bEnergyRemote;
    zbmap8   u8ScheduleMode;
} tsCLD_PP;
```

where:

- `u8TotalProfileNum` is the number of power profiles supported by the device (must be between 1 and 254, inclusive)

- `bMultipleScheduling` is a boolean indicating whether the server side of the cluster supports the scheduling of multiple energy phases or just a single energy phase at a time (according to commands received from the client):
  - TRUE if multiple energy phase scheduling is possible
  - FALSE if only single energy phase scheduling is possible
- `u8EnergyFormatting` indicates the format of the Energy fields in the Power Profile Notification and Power Profile Response:
  - Bits 0-2: Number of digits to the right of the decimal point
  - Bits 3-6: Number of digits to the left of the decimal point
  - Bit 7: If set to '1', any leading zeros will be removed
- `bEnergyRemote` is a boolean indicating whether the cluster server (appliance) is configured for remote control (of energy management):
  - TRUE if at least one power profile is enabled for remote control
  - FALSE if no power profile is enabled for remote control

  This attribute is linked to the `bPowerProfileRemoteControl` field in the power profile record (see Section 7.9.13) - if the latter field is set to TRUE, the attribute is also automatically set to TRUE.
- `u8ScheduleMode` indicates the criterion (cheapest or greenest) that should be used by the cluster client (e.g. energy management system) to schedule the power profiles:
  - 0x00 - criterion is left to the cluster server to choose
  - 0x01 - cheapest mode (minimise cost of energy usage)
  - 0x02 - greenest mode (maximise use of renewable energy sources)
  - 0x03 - compromise between cheapest and greenest

  All other values are reserved.

7.3 Power Profiles

An appliance can have one or more power profiles. An example of an appliance with multiple power profiles is a washing machine which has a number of programmes for different types of materials and loads.

Note: The number of power profiles on a device must be defined in the file **zcl_options.h** (see Section 7.10). *However, in the current HA release, a device is restricted to having only one power profile.*

An individual power profile comprises a series of energy phases with different power demands. For example, these phases may correspond to the different cycles of a washing machine programme, such as wash, rinse, spin. Details of a power profile, including these energy phases, are held in an entry of the power profile table on the cluster server (appliance).

If the appliance is to be remotely controlled, the controller (cluster client) must 'learn' the details of the appliance's power profile so that it can control the scheduling of the energy phases. The schedule of a power profile is decided by the client, and includes energy phases and their relative start-times (the energy phases are not necessarily contiguous in time). A schedule is illustrated in Figure 4. The client must communicate the schedule for a power profile to the server where the schedule will be executed.

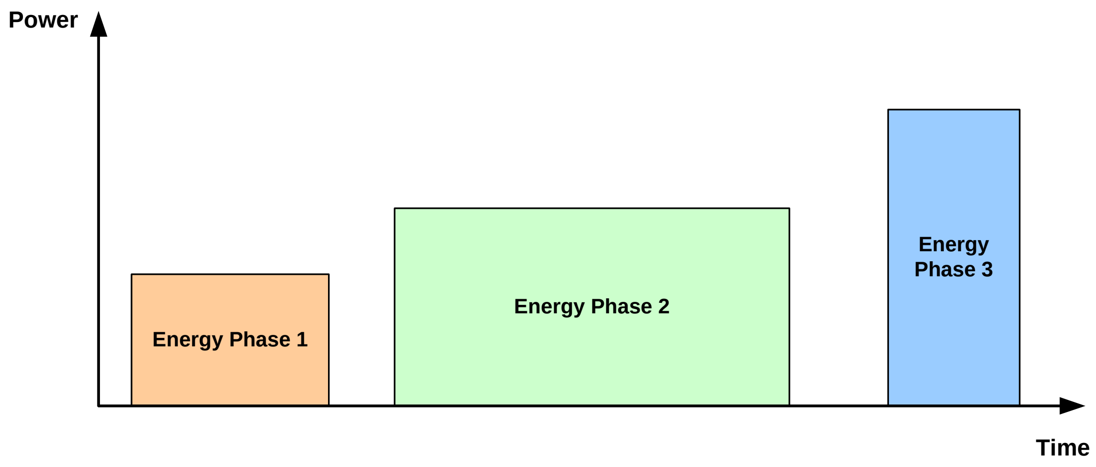


Figure 4: Schedule of Energy Phases of a Power Profile

7.4 Power Profile Operations

This section describes the main operations to be performed on the Power Profile cluster server (appliance) and client (controller).

7.4.1 Initialisation

The Power Profile cluster must be initialised on both the cluster server and client. This can be done using the function **eCLD_PPCreatePowerProfile()**, which creates an instance of the Power Profile cluster on a local endpoint.

If you are using a ZigBee device which includes the Power Profile cluster, the above function will be automatically called by the initialisation function for the device. You only need to call **eCLD_PPCreatePowerProfile()** explicitly when setting up a custom endpoint containing one or more selected clusters (rather than the whole set of clusters supported by a standard ZigBee device).

7.4.2 Adding and Removing a Power Profile (Server Only)

A Power Profile cluster server (appliance) will support one or more power profiles. Information on these power profiles is held on the server in a power profile table, where each table entry contains information on one supported power profile.

Note: In the current HA release, a device is restricted to having only one power profile.

The application on the appliance can perform various operations on the power profile table, as described in the sub-sections below.

7.4.2.1 Adding a Power Profile Entry

The server application can introduce a new power profile by adding a corresponding entry to the power profile table using the function **eCLD_PPAddPowerProfileEntry()**. The new power profile table entry is specified in a `tsCLD_PPEntry` structure (see Section 7.9.2) supplied to this function. This structure includes the Power Profile ID - these identifiers should be numbered consecutively from 1 to 255.

The function **eCLD_PPAddPowerProfileEntry()** can also be used to replace (over-write) an existing power profile table entry, in which case the new entry should have the same Power Profile ID as the existing entry to be replaced.

7.4.2.2 Removing a Power Profile Entry

The server application can remove a power profile from the device by calling the function **eCLD_PPRemovePowerProfileEntry()** to delete the corresponding entry of the local power profile table. The entry to be deleted is specified by means of the relevant Power Profile ID.

7.4.2.3 Obtaining a Power Profile Entry

The server application can obtain the details of a power profile supported by the server by reading the corresponding entry of the power profile table using the function **eCLD_PPGetPowerProfileEntry()**. The required entry is specified by means of the relevant Power Profile ID.

7.4.3 Communicating Power Profiles

In order to control the power consumption of the appliance (by scheduling the energy phases of the power profile), the controller (cluster client) must 'learn' the power profiles supported by the appliance (server). This may be done through requests or notifications, as described in the sub-sections below.

Note: In order remotely control the appliance from a controller for energy management, the attribute `bEnergyRemote` of the Power Profile cluster on the server device must be set to TRUE (see Section 7.2).

7.4.3.1 Requesting a Power Profile (by Client)

In order to 'learn' a power profile supported by the server, the client application can request this profile from the server by calling the **eCLD_PPPowerProfileReqSend()** function, which sends a Power Profile Request to the server. This function can be used to request a specific power profile (specified using its Power Profile ID) or all the power profiles supported by the server.

On receiving the server's response, an E_CLD_PP_CMD_POWER_PROFILE_RSP event is generated on the client for each energy phase within the power profile. The reported information is contained in a `tsCLD_PP_PowerProfilePayload` structure (see Section 7.9.4). The application may store or discard this information, as required. By receiving the energy phase information in individual events, the application only needs to use as much memory as is required to store the relevant energy phase data.

Note: The client application may first use the function **eCLD_PPPowerProfileStateReqSend()** to request the identifiers of the power profiles that are currently supported on the server.

7.4.3.2 Notification of a Power Profile (by Server)

The cluster server may send unsolicited notifications of the power profiles that it supports to the client. To do this, the server application must call the function **eCLD_PPPowerProfileNotificationSend()** which sends a Power Profile Notification containing the essential details of one supported power profile (such as the energy phases within the profile). This information is supplied to the function in a `tsCLD_PP_PowerProfilePayload` structure (see Section 7.9.4). If the server supports multiple power profiles, a separate notification must be sent for each profile.

On receiving the notification on the client, the event E_CLD_PP_CMD_POWER_PROFILE_NOTIFICATION is generated on the client for each energy phase within the power profile. The reported information is contained in a `tsCLD_PP_PowerProfilePayload` structure (see Section 7.9.4). The application may store or discard this information, as required. By receiving the energy phase information in individual events, the application only needs to use as much memory as is required to store the relevant energy phase data.

7.4.4 Communicating Schedule Information

A power profile schedule comprises a sequence of energy phases and their relative start-times (the energy phases may have gaps between them):

- An energy phase is identified by its Energy Phase Identifier, in the range 1 to 255 (inclusive)
- The start-time of an energy phase is expressed as a delay, in minutes, from the end of the previous energy phase. For the first energy phase of a power profile schedule, this delay is measured from the time that the schedule was started

Note: The normal duration of an energy phase, in minutes, is fixed and is specified in the energy phase information in the power profile.

Although a power profile on the cluster server may support multiple energy phases, the schedule for the power profile may possibly incorporate only a sub-set of these phases. The set of energy phases in a schedule is chosen by the client (controller), which must communicate this schedule to the server (appliance). This may be done through a request or notification, as described in Section 7.4.4.1 and Section 7.4.4.2 below.

7.4.4.1 Requesting a Schedule (by Server)

The server application can request a schedule for a supported power profile from the client by calling the function **eCLD_PPEnergyPhasesScheduleReqSend()**, which sends an Energy Phases Schedule Request to the client.

The client can only return the requested schedule information if it stores this type of information for the power profile. If this is the case, an E_CLD_PP_CMD_ENERGY_PHASES_SCHEDULE_REQ event will be generated on the client, with the `bIsInfoAvailable` field set to TRUE in the event structure `tsCLD_PPCallBackMessage`, and the client will send an Energy Phases Schedule Response back to the server. Otherwise, the client will send a ZCL default response with status NOT_FOUND.

On receiving an Energy Phases Schedule Response from the client, the event E_CLD_PP_CMD_ENERGY_PHASES_SCHEDULE_RSP is generated on the server, containing the requested schedule information in a `tsCLD_PP_EnergyPhasesSchedulePayload` structure (see Section 7.9.6). One or more of the following outcomes will result:

- If the attribute `bEnergyRemote` is set to FALSE on the server (no remote control of the device), the server will simply reject the received schedule.
- If the received schedule information contains an `u16MaxActivationDelay` value of zero for an energy phase (see Section 7.9.11), this energy phase will be rejected by the server although other valid energy phases will be accepted. For each rejected energy phase, the server will send a ZCL default response with status NOT_AUTHORIZED to the client.
- If the received schedule information results in an update of the power profile schedule on the server, the server will automatically send an Energy Phases Schedule State Notification back to the client. On receiving this notification, an E_CLD_PP_CMD_ENERGY_PHASES_SCHEDULE_STATE_NOTIFICATION event will be generated on the client.

Note: Before requesting a power profile schedule, the server application may send the schedule's timing constraints to the client using the function **eCLD_PPPowerProfileScheduleConstraintsNotificationSend()**. The client application can alternatively request these schedule constraints from the server by calling **eCLD_PPPowerProfileScheduleConstraintsReqSend()**.

7.4.4.2 Notification of a Schedule (by Client)

The cluster client may send an unsolicited notification of a power profile schedule to the server. To do this, the client application must call the function **eCLD_PPEnergyPhasesScheduleNotificationSend()** which sends an Energy Phases Schedule Notification containing the schedule. This information is supplied to the function in a `tsCLD_PP_EnergyPhasesSchedulePayload` structure (see Section 7.9.6).

On receiving the notification on the server, the event E_CLD_PP_CMD_ENERGY_PHASES_SCHEDULE_NOTIFICATION is generated, containing the sent power profile schedule. One or more of the following outcomes will result:

- If the attribute `bEnergyRemote` is set to FALSE on the server (no remote control of the device), the server will simply reject the received schedule.
- If the received schedule information contains an `u16MaxActivationDelay` value of zero for an energy phase (see Section 7.9.11), this energy phase will be rejected by the server although other valid energy phases will be accepted. For each rejected energy phase, the server will send a ZCL default response with status NOT_AUTHORIZED to the client.
- If the received schedule information results in an update of the power profile schedule on the server, the server will automatically send an Energy Phases Schedule State Notification back to the client. On receiving this notification, an E_CLD_PP_CMD_ENERGY_PHASES_SCHEDULE_STATE_NOTIFICATION event will be generated on the client.

7.4.4.3 Notification of Energy Phases in Power Profile Schedule (by Server)

The server application can use the function **eCLD_PPEnergyPhasesScheduleStateNotificationSend()** to send an unsolicited Energy Phases Schedule State Notification to a cluster client, in order to inform the client of the energy phases that are in the schedule of a particular power profile.

7.4.4.4 Requesting the Scheduled Energy Phases (by Client)

The client application can use the function **eCLD_PPEnergyPhasesScheduleStateReqSend()** to send an Energy Phases Schedule State Request to the cluster server, in order to obtain the schedule of energy phases for a particular power profile on the server.

On receiving the response on the client, the event E_CLD_PP_CMD_ENERGY_PHASES_SCHEDULE_STATE_RSP is generated, containing the requested schedule information. The obtained schedule can be used to re-align the schedule information on the client with the information on the server - for example, after a reset of the client device.

7.4.5 Executing a Power Profile Schedule

After receiving a power profile schedule from the client (as described in Section 7.4.4), the server can start execution of the schedule. The instruction to start the schedule comes from the client in the form of an Energy Phases Schedule Notification. To issue this instruction, the client application must call the function **eCLD_PPEnergyPhasesScheduleNotificationSend()**. On receiving the notification, the server will automatically start the schedule.

The possible states of a power profile are fully detailed in Section 7.8.2 but, generally, it will move through the following principal states before, during and after execution:

- E_CLD_PP_STATE_PROGRAMMED: The power profile is defined in the local power profile table but a schedule has not been received from the client. Even without a schedule from the client, a schedule of energy phases that was defined when the power profile was introduced using the function **eCLD_PPAddPowerProfileEntry()** can be started from this state (see below).
- E_CLD_PP_STATE_WAITING_TO_START: The power profile will be in this state before the first energy phase starts and between energy phases (provided there is a gap between the end of one phase and the beginning of the next).
- E_CLD_PP_STATE_RUNNING: An energy phase is running.
- E_CLD_PP_STATE_ENDED: The final energy phase has completed.

Once a schedule has started, the server application must progress execution through the different states of the schedule by periodically calling the function **eCLD_PPSchedule()** once per second. This function will move the power profile to the next state, if it is due to start, and update the relevant state and timing parameters.

Note: The server application can also use the function **eCLD_PPSetPowerProfileState()** to 'manually' move execution of the schedule to a particular (valid) state, irrespective of whether the target state is scheduled. This function can be used by the server application to locally start a schedule from the 'programmed' state.

Whenever there is a change of state of a power profile, the cluster server will automatically send a Power Profile State Notification to the client (the server application can also send such a notification 'manually' by calling the function **eCLD_PPPowerProfileStateNotificationSend()**). The notification contains a power profile record which specifies the active power profile, the energy phase that is currently running (or due to run next) and the current state of the power profile. These notifications allow the controller to monitor the appliance. On receiving a notification on the client, an E_CLD_PP_CMD_POWER_PROFILE_STATE_NOTIFICATION event is generated, containing the sent power profile state information in a `tsCLD_PP_PowerProfileStatePayload` structure (see Section 7.9.5).

7.4.6 Communicating Price Information

The cost of implementing a power profile schedule on an appliance (cluster server) is determined/calculated by the controller (cluster client). The server can request price information from the client in a number of ways, as described below.

Note: Use of the Power Profile Price functions, referenced below, must be enabled in the compile-time options, as described in Section 7.10.

7.4.6.1 Requesting Cost of a Power Profile Schedule (by Server)

The server application can use the function **eCLD_PPGetPowerProfilePriceSend()** to send a Get Power Profile Price Request to the client, in order to request the cost of executing the schedule of a particular power profile.

The client can only return the requested information if price-related information about the power profile is held on the client device. If this is the case, an E_CLD_PP_CMD_GET_POWER_PROFILE_PRICE event will be generated on the client, with the `bIsInfoAvailable` field set to TRUE in the event structure `tsCLD_PPCallBackMessage` and the client will send a Get Power Profile Price Response back to the server. Otherwise, the client will send a ZCL default response with status NOT_FOUND.

On receiving a Get Power Profile Price Response on the server, the event E_CLD_PP_CMD_GET_POWER_PROFILE_PRICE_RSP is generated, containing the requested price information (if available).

Alternatively, the server application can use the function **eCLD_PPGetPowerProfilePriceExtendedSend()** to send a Get Power Profile Price Extended Request to a cluster client, in order to request specific cost information about a power profile supported by the server. The cost of executing a power profile can be requested with either scheduled energy phases or contiguous energy phases (no gaps between them). This request will be handled by the client as described above for an ordinary Get Power Profile Price Request. However, the response will result in an E_CLD_PP_CMD_GET_POWER_PROFILE_PRICE_EXTENDED_RSP event on the server, containing the requested price information (if available).

7.4.6.2 Requesting Cost of Power Profile Schedules Over a Day (by Server)

The server application can use the **eCLD_PPGetOverallSchedulePriceSend()** function to send a Get Overall Schedule Price Request to the client, in order to obtain the overall cost of all the power profiles that will be executed over the next 24 hours.

The client can only return the requested information if price-related information about the relevant power profiles is held on the client device. If this is the case, an E_CLD_PP_CMD_GET_OVERALL_SCHEDULE_PRICE event will be generated on the client, with the `bIsInfoAvailable` field set to TRUE in the event structure `tsCLD_PPCallBackMessage`. Otherwise, the client will generate a ZCL default response with status NOT_FOUND.

On receiving a Get Overall Schedule Price Response on the server, the event E_CLD_PP_CMD_GET_OVERALL_SCHEDULE_PRICE_RSP is generated, containing the requested price information (if available).

7.5 Power Profile Events

The Power Profile cluster has its own events that are handled through the callback mechanism outlined in Section 4.7 (and fully detailed in the *ZCL User Guide (JN-UG-3077)*). The cluster contains its own event handler. However, if a device uses this cluster then application-specific Power Profile event handling must be included in the user-defined callback function for the associated endpoint, where this callback function is registered through the relevant endpoint registration function. This callback function will then be invoked when a Power Profile event occurs and needs the attention of the application.

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

```
typedef struct
{
    uint8           u8CommandId;
#ifdef PP_CLIENT
    bool            bIsInfoAvailable;
#endif
    union
    {
    tsCLD_PP_PowerProfileReqPayload *psPowerProfileReqPayload;
    tsCLD_PP_GetPowerProfilePriceExtendedPayload
                                *psGetPowerProfilePriceExtendedPayload;
    } uReqMessage;
    union
    {
    tsCLD_PP_GetPowerProfilePriceRspPayload *psGetPowerProfilePriceRspPayload;
    tsCLD_PP_GetOverallSchedulePriceRspPayload
                                        *psGetOverallSchedulePriceRspPayload;
    tsCLD_PP_EnergyPhasesSchedulePayload    *psEnergyPhasesSchedulePayload;
    tsCLD_PP_PowerProfileScheduleConstraintsPayload
                                   *psPowerProfileScheduleConstraintsPayload;
    tsCLD_PP_PowerProfilePayload            *psPowerProfilePayload;
    tsCLD_PP_PowerProfileStatePayload       *psPowerProfileStatePayload;
    }uRespMessage;
} tsCLD_PPCallBackMessage;
```

The above structure is fully described in Section 7.9.1.

When a Power Profile event occurs, one of the command types listed in Table 35 and Table 36 is specified through the `u8CommandId` field of the `tsCLD_PPCallBackMessage` structure. This command type determines which command payload is used from the unions `uReqMessage` (for request commands) and `uRespMessage` (for response and notification commands).

u8CommandId Enumeration	Description/Payload Type
E_CLD_PP_CMD_POWER_PROFILE_REQ	A Power Profile Request has been received by the server (appliance). **tsCLD_PP_PowerProfileReqPayload**
E_CLD_PP_CMD_POWER_PROFILE_STATE_REQ	A Power Profile State Request has been received by the server (appliance).
E_CLD_PP_CMD_GET_POWER_PROFILE_PRICE_RSP	A Get Power Profile Price Response has been received by the server (appliance), following a previously sent Get Power Profile Price Request. **tsCLD_PP_GetPowerProfilePriceRspPayload**
E_CLD_PP_CMD_GET_POWER_PROFILE_PRICE_EXTENDED_RSP	A Get Power Profile Price Extended Response has been received by the server (appliance), following a previously sent Get Power Profile Price Extended Request. **tsCLD_PP_GetPowerProfilePriceRspPayload**
E_CLD_PP_CMD_GET_OVERALL_SCHEDULE_PRICE_RSP	A Get Overall Schedule Price Response has been received by the server (appliance), following a previously sent Get Overall Schedule Price Request. **tsCLD_PP_GetOverallSchedulePriceRspPayload**
E_CLD_PP_CMD_ENERGY_PHASES_SCHEDULE_NOTIFICATION	An Energy Phases Schedule Notification has been received by the server (appliance). **tsCLD_PP_EnergyPhasesSchedulePayload**
E_CLD_PP_CMD_ENERGY_PHASES_SCHEDULE_RSP	An Energy Phases Schedule Response has been received by the server (appliance), following a previously sent Energy Phases Schedule Request. **tsCLD_PP_EnergyPhasesSchedulePayload**
E_CLD_PP_CMD_ENERGY_PHASES_SCHEDULE_STATE_RSP	An Energy Phases Schedule State Response has been received by the server (appliance), following a previously sent Energy Phases Schedule State Request. **tsCLD_PP_EnergyPhasesSchedulePayload**
E_CLD_PP_CMD_GET_POWER_PROFILE_SCHEDULE_CONSTRAINTS_REQ	A Get Power Profile Schedule Constraints Request has been received by the server (appliance). **tsCLD_PP_PowerProfileReqPayload**
E_CLD_PP_CMD_ENERGY_PHASES_SCHEDULE_STATE_REQ	An Energy Phases Schedule State Request has been received by the server (appliance). **tsCLD_PP_PowerProfileReqPayload**

Table 35: Power Profile Command Types (Events on Server)

u8CommandId Enumeration	Description/Payload Type
E_CLD_PP_CMD_POWER_PROFILE_NOTIFICATION	A Power Profile Notification has been received by the client (controller). **tsCLD_PP_PowerProfilePayload**
E_CLD_PP_CMD_POWER_PROFILE_STATE_ NOTIFICATION	A Power Profile State Notification has been received by the client (controller). **tsCLD_PP_PowerProfileStatePayload**
E_CLD_PP_CMD_POWER_PROFILE_RSP	A Power Profile Response has been received by the client (controller), following a previously sent Power Profile Request. **tsCLD_PP_PowerProfilePayload**
E_CLD_PP_CMD_POWER_PROFILE_STATE_RSP	A Power Profile State Response has been received by the client (controller), following a previously sent Power Profile State Request. **tsCLD_PP_PowerProfileStatePayload**
E_CLD_PP_CMD_GET_POWER_PROFILE_PRICE	A Get Power Profile Price Request has been received by the client (controller). **tsCLD_PP_PowerProfileReqPayload**
E_CLD_PP_CMD_GET_OVERALL_SCHEDULE_ PRICE	A Get Overall Schedule Price Request has been received by the client (controller).
E_CLD_PP_CMD_ENERGY_PHASES_SCHEDULE_ REQ	An Energy Phases Schedule Request has been received by the client (controller). **tsCLD_PP_PowerProfileReqPayload**
E_CLD_PP_CMD_ENERGY_PHASES_SCHEDULE_ STATE_NOTIFICATION	An Energy Phases Schedule State Notification has been received by the client (controller). **tsCLD_PP_EnergyPhasesSchedulePayload**
E_CLD_PP_CMD_SCHEDULE_CONSTRAINTS_ NOTIFICATION	A Power Profile Schedule Constraints Notification has been received by the client (controller). **tsCLD_PP_PowerProfileScheduleConstraintsPayload**
E_CLD_PP_CMD_GET_POWER_PROFILE_ SCHEDULE_CONSTRAINTS_RSP	A Power Profile Schedule Constraints Response has been received by the client (controller). **tsCLD_PP_PowerProfileScheduleConstraintsPayload**
E_CLD_PP_CMD_GET_POWER_PROFILE_PRICE_ EXTENDED	A Get Power Profile Price Extended Request has been received by the client (controller). **tsCLD_PP_GetPowerProfilePriceExtendedPayload**

Table 36: Power Profile Command Types (Events on Client)

7.6 Functions

The Power Profile cluster functions provided in the HA API are described in the following three sub-sections, according to the side(s) of the cluster on which they can be used:

- Server/client function are described in Section 7.6.1
- Server functions are described in Section 7.6.2
- Client functions are described in Section 7.6.3

7.6.1 Server/Client Function

The following Power Profile cluster function is provided in the HA API and can be used on either a cluster server or cluster client:

Function	Page
eCLD_PPCreatePowerProfile	115

eCLD_PPCreatePowerProfile

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

Description

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

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

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

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

The function requires an array to be declared for internal use, which contains one element (of type **uint8**) for each attribute of the cluster. The array length should therefore equate to the number of attributes supported by the Power Profile cluster, which is 5.

The array declaration should be as follows:

```
uint8 au8AppPPClusterAttributeControlBits[5];
```

The function will initialise the array elements to zero.

Parameters

psClusterInstance	Pointer to structure containing information about the cluster instance to be created (see the *ZCL User Guide (JN-UG-3077)*). This structure will be updated by the function by initialising individual structure fields.
bIsServer	Type of cluster instance (server or client) to be created: TRUE - server FALSE - client
psClusterDefinition	Pointer to structure indicating the type of cluster to be created (see the *ZCL User Guide (JN-UG-3077)*). In this case, this structure must contain the details of the Power Profile cluster. This parameter can refer to a pre-filled structure called `sCLD_PP` which is provided in the **PowerProfile.h** file.
pvEndPointSharedStructPtr	Pointer to the shared structure used for attribute storage. This parameter should be the address of the structure of type `tsCLD_PP` which defines the attributes of the Power Profile cluster. The function will initialise the attributes with default values.
pu8AttributeControlBits	Pointer to an array of **uint8** values, with one element for each attribute in the cluster (see above).
psCustomDataStructure	Pointer to a structure containing the storage for internal functions of the cluster (see Section 7.9.14)

Returns

E_ZCL_SUCCESS

E_ZCL_FAIL

E_ZCL_ERR_PARAMETER_NULL

E_ZCL_ERR_INVALID_VALUE

7.6.2 Server Functions

The following Power Profile cluster functions are provided in the HA API and can be used on a cluster server only:

Function	Page
eCLD_PPSchedule	118
eCLD_PPSetPowerProfileState	119
eCLD_PPAddPowerProfileEntry	120
eCLD_PPRemovePowerProfileEntry	121
eCLD_PPGetPowerProfileEntry	122
eCLD_PPPowerProfileNotificationSend	123
eCLD_PPEnergyPhaseScheduleStateNotificationSend	124
eCLD_PPPowerProfileScheduleConstraintsNotificationSend	125
eCLD_PPEnergyPhasesScheduleReqSend	127
eCLD_PPPowerProfileStateNotificationSend	128
eCLD_PPGetPowerProfilePriceSend	129
eCLD_PPGetPowerProfilePriceExtendedSend	130
eCLD_PPGetOverallSchedulePriceSend	132

eCLD_PPSchedule

```
teZCL_Status eCLD_PPSchedule(void);
```

Description

This function can be used on a cluster server to update the state of the currently active power profile and the timings required for scheduling. When called, the function automatically makes any required changes according to the scheduled energy phases for the power profile. If no change is scheduled, the function only updates timing information. If a change is required, it also updates the power profile state and the ID of the energy phase currently being executed.

The function should be called once per second to progress the active power profile schedule and the application should provide a 1-second timer to prompt these function calls.

Parameters

None

Returns

E_ZCL_SUCCESS

E_ZCL_FAIL

E_ZCL_ERR_PARAMETER_NULL

E_ZCL_ERR_INVALID_VALUE

eCLD_PPSetPowerProfileState

```
teZCL_CommandStatus eCLD_PPSetPowerProfileState(
            uint8 u8SourceEndPointId,
            uint8 u8PowerProfileId,
            teCLD_PP_PowerProfileState sPowerProfileState);
```

Description

This function can be used on a cluster server to move the specified power profile to the specified target state. Enumerations for the possible states are provided, and are listed and described in Section 7.8.2.

The function performs the following checks:

- Checks whether the specified Power Profile ID exists (if not, the function returns with the status E_ZCL_CMDS_NOT_FOUND)
- Checks whether the specified target state is a valid state (if not, the function returns with the status E_ZCL_CMDS_INVALID_VALUE)
- Checks whether the power profile is currently able move to the target state (if not, the function returns with the status E_ZCL_CMDS_INVALID_FIELD)

Note: The power profile state can be changed by this function only if the move from the existing state to the target state is a valid change according to the HA specification.

If all the checks are successful, the move is implemented (and the function returns with the status E_ZCL_CMD_SUCCESS).

Parameters

u8SourceEndPointId	Number of local endpoint on which cluster resides
u8PowerProfileId	Identifier of the power profile
sPowerProfileState	Target state to which power profile is to be moved - enumerations are provided (see Section 7.8.2)

Returns

E_ZCL_CMD_SUCCESS

E_ZCL_CMDS_NOT_FOUND

E_ZCL_CMDS_INVALID_VALUE

E_ZCL_CMDS_INVALID_FIELD

eCLD_PPAddPowerProfileEntry

```
teZCL_Status eCLD_PPAddPowerProfileEntry(
                    uint8 u8SourceEndPointId,
                    tsCLD_PPEntry *psPowerProfileEntry);
```

Description

This function can be used on a cluster server to introduce a new power profile by adding an entry to the local power profile table.

The function checks whether there is sufficient space in the table for the new power profile entry (if not, the function returns with the status E_ZCL_ERR_INSUFFICIENT_SPACE).

An existing power profile entry can be over-written with a new profile by specifying the same Power Profile ID (in the new entry structure).

The function will also update two of the cluster attributes (if needed), as follows.

- If a power profile is introduced which has multiple energy phases (as indicated by the `u8NumOfScheduledEnergyPhases` field of the `tsCLD_PPEntry` structure), the attribute `bMultipleScheduling` will be set to TRUE (if not already TRUE)
- If a power profile is introduced which allows remote control for energy management (as indicated by the `bPowerProfileRemoteControl` field of the `tsCLD_PPEntry` structure), the attribute `bEnergyRemote` will be set to TRUE (if not already TRUE)

Parameters

u8SourceEndPointId	Number of local endpoint on which cluster resides
psPowerProfileEntry	Structure containing the power profile to add (see Section 7.9.2)

Returns

E_ZCL_SUCCESS
E_ZCL_FAIL
E_ZCL_ERR_PARAMETER_NULL
E_ZCL_ERR_INSUFFICIENT_SPACE

eCLD_PPRemovePowerProfileEntry

```
teZCL_Status eCLD_PPRemovePowerProfileEntry(
                                uint8 u8SourceEndPointId,
                                uint8 u8PowerProfileId);
```

Description

This function can be used on a cluster server to remove a power profile by deleting the relevant entry in the local power profile table.

Parameters

u8SourceEndPointId	Number of local endpoint on which cluster resides
u8PowerProfileId	Identifier of power profile to be removed

Returns

E_ZCL_SUCCESS

E_ZCL_FAIL

eCLD_PPGetPowerProfileEntry

```
teZCL_Status eCLD_PPGetPowerProfileEntry(
                    uint8 u8SourceEndPointId,
                    uint8 u8PowerProfileId,
                    tsCLD_PPEntry **ppsPowerProfileEntry);
```

Description

This function can be used on a cluster server to obtain an entry from the local power profile table. The required entry is specified using the relevant Power Profile ID. The function obtains a pointer to the relevant entry, if it exists - a pointer must be provided to a location to receive the pointer to the entry.

If no entry with the specified Power Profile ID is found, the function returns E_ZCL_ERR_INVALID_VALUE.

Parameters

u8SourceEndPointId	Number of local endpoint on which cluster resides
u8PowerProfileId	Identifier of power profile to be obtained
ppsPowerProfileEntry	Pointer to a location to receive a pointer to the required power profile table entry (see Section 7.9.2)

Returns

E_ZCL_SUCCESS
E_ZCL_FAIL
E_ZCL_ERR_INVALID_VALUE

eCLD_PPPowerProfileNotificationSend

```
teZCL_Status eCLD_PPPowerProfileNotificationSend(
            uint8 u8SourceEndPointId,
            uint8 u8DestinationEndPointId,
            tsZCL_Address *psDestinationAddress,
            uint8 *pu8TransactionSequenceNumber,
            tsCLD_PP_PowerProfilePayload *psPayload);
```

Description

This function can be used on the cluster server to send a Power Profile Notification to a cluster client, in order to inform the client about one power profile supported by the server. The notification contains essential information about the power profile, including the energy phases supported by the profile (and certain details about them). If the server supports multiple power profiles, the function must be called for each profile separately.

On receiving the notification on the client, the event E_CLD_PP_CMD_POWER_PROFILE_NOTIFICATION will be generated, containing the sent power profile information in a `tsCLD_PP_PowerProfilePayload` structure (see Section 7.9.4).

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

Parameters

u8SourceEndPointId	Number of local endpoint on which cluster server resides
u8DestinationEndPointId	Number of remote endpoint on which cluster client resides
psDestinationAddress	Pointer to a structure containing the destination address of the client node
pu8TransactionSequenceNumber	Pointer to a location to receive the Transaction Sequence Number (TSN) of the message
psPayload	Pointer to structure containing the payload for the request (see Section 7.9.4), including essential information about the power profile

Returns

E_ZCL_SUCCESS

E_ZCL_FAIL

E_ZCL_ERR_EP_UNKNOWN

E_ZCL_ERR_ZBUFFER_FAIL

E_ZCL_ERR_ZTRANSMIT_FAIL

eCLD_PPEnergyPhaseScheduleStateNotificationSend

```
teZCL_Status
eCLD_PPEnergyPhasesScheduleStateNotificationSend(
    uint8 u8SourceEndPointId,
    uint8 u8DestinationEndPointId,
    tsZCL_Address *psDestinationAddress,
    uint8 *pu8TransactionSequenceNumber,
    tsCLD_PP_EnergyPhasesSchedulePayload
                                        *psPayload);
```

Description

This function can be used on the cluster server to send an Energy Phases Schedule State Notification to a cluster client, in order to inform the client of the energy phases that are in the schedule of a particular power profile. The function is used to send an unsolicited command.

On receiving the notification on the client, the event E_CLD_PP_CMD_ENERGY_PHASES_SCHEDULE_STATE_NOTIFICATION will be generated, containing the sent power profile information in a `tsCLD_PP_EnergyPhasesSchedulePayload` structure (see Section 7.9.6).

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

Parameters

u8SourceEndPointId	Number of local endpoint on which cluster server resides
u8DestinationEndPointId	Number of remote endpoint on which cluster client resides
psDestinationAddress	Pointer to a structure containing the destination address of the client node
pu8TransactionSequenceNumber	Pointer to a location to receive the Transaction Sequence Number (TSN) of the message
psPayload	Pointer to structure containing the payload for the request (see Section 7.9.6), including the identifier of the relevant power profile and the associated schedule of energy phases

Returns

E_ZCL_SUCCESS
E_ZCL_FAIL
E_ZCL_ERR_EP_UNKNOWN
E_ZCL_ERR_ZBUFFER_FAIL
E_ZCL_ERR_ZTRANSMIT_FAIL

eCLD_PPPowerProfileScheduleConstraintsNotificationSend

```
teZCL_Status
eCLD_PPPowerProfileScheduleConstraintsNotificationSend(
    uint8 u8SourceEndPointId,
    uint8 u8DestinationEndPointId,
    tsZCL_Address *psDestinationAddress,
    uint8 *pu8TransactionSequenceNumber,
    tsCLD_PP_PowerProfileScheduleConstraintsPayload
                                              *psPayload);
```

Description

This function can be used on the cluster server to send a Power Profile Schedule Constraints Notification to a cluster client, in order to inform the client of the schedule restrictions on a particular power profile. The constraints are specified in a `tsCLD_PP_PowerProfileScheduleConstraintsPayload` structure (see Section 7.9.7). They can subsequently be used by the client in calculating the schedule for the energy phases of the power profile. The function is used to send an unsolicited command.

On receiving the notification on the client, the event E_CLD_PP_CMD_SCHEDULE_CONSTRAINTS_NOTIFICATION will be generated, containing the sent power profile constraint information in a `tsCLD_PP_PowerProfileScheduleConstraintsPayload` structure.

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

Parameters

u8SourceEndPointId	Number of local endpoint on which cluster server resides
u8DestinationEndPointId	Number of remote endpoint on which cluster client resides
psDestinationAddress	Pointer to a structure containing the destination address of the client node
pu8TransactionSequenceNumber	Pointer to a location to receive the Transaction Sequence Number (TSN) of the message
psPayload	Pointer to structure containing the payload for the request (see Section 7.9.7), including the identifier of the relevant power profile and the associated schedule constraints

Returns

E_ZCL_SUCCESS

E_ZCL_FAIL

E_ZCL_ERR_EP_UNKNOWN

E_ZCL_ERR_ZBUFFER_FAIL

E_ZCL_ERR_ZTRANSMIT_FAIL

eCLD_PPEnergyPhasesScheduleReqSend

teZCL_Status eCLD_PPEnergyPhasesScheduleReqSend(
uint8 *u8SourceEndPointId*,
uint8 *u8DestinationEndPointId*,
tsZCL_Address **psDestinationAddress*,
uint8 **pu8TransactionSequenceNumber*,
tsCLD_PP_PowerProfileReqPayload **psPayload*);

Description

This function can be used on the cluster server to send an Energy Phases Schedule Request to a cluster client, in order to obtain the schedule of energy phases for a particular power profile.

The function is non-blocking and will return immediately. On successfully receiving an Energy Phases Schedule Response from the client, an E_CLD_PP_CMD_ENERGY_PHASE_SCHEDULE_RSP event will be generated on the server, containing the requested schedule information in a `tsCLD_PP_EnergyPhasesSchedulePayload` structure (see Section 7.9.6). For full details of handling an Energy Phases Schedule Request, refer to Section 7.4.4.1.

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

Parameters

u8SourceEndPointId	Number of local endpoint on which cluster server resides
u8DestinationEndPointId	Number of remote endpoint on which cluster client resides
psDestinationAddress	Pointer to a structure containing the destination address of the client node
pu8TransactionSequenceNumber	Pointer to a location to receive the Transaction Sequence Number (TSN) of the message
psPayload	Pointer to structure containing the payload for the request (see Section 7.9.3), including the identifier of the relevant power profile

Returns

E_ZCL_SUCCESS

E_ZCL_FAIL

E_ZCL_ERR_EP_UNKNOWN

E_ZCL_ERR_ZBUFFER_FAIL

E_ZCL_ERR_ZTRANSMIT_FAIL

eCLD_PPPowerProfileStateNotificationSend

```
teZCL_Status eCLD_PPPowerProfileStateNotificationSend(
    uint8 u8SourceEndPointId,
    uint8 u8DestinationEndPointId,
    tsZCL_Address *psDestinationAddress,
    uint8 *pu8TransactionSequenceNumber,
    tsCLD_PP_PowerProfileStatePayload *psPayload);
```

Description

This function can be used on the cluster server to send a Power Profile State Notification to a cluster client, in order to inform the client of the state of the power profile that is currently active on the server. The function is used to send an unsolicited command.

On receiving the notification on the client, the event E_CLD_PP_CMD_POWER_PROFILE_STATE_NOTIFICATION will be generated, containing the sent power profile state information in a `tsCLD_PP_PowerProfileStatePayload` structure (see Section 7.9.5).

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

Parameters

u8SourceEndPointId	Number of local endpoint on which cluster server resides
u8DestinationEndPointId	Number of remote endpoint on which cluster client resides
psDestinationAddress	Pointer to a structure containing the destination address of the client node
pu8TransactionSequenceNumber	Pointer to a location to receive the Transaction Sequence Number (TSN) of the message
psPayload	Pointer to structure containing the payload for the command (see Section 7.9.5), including the identifier and state of the relevant power profile

Returns

E_ZCL_SUCCESS

E_ZCL_FAIL

E_ZCL_ERR_EP_UNKNOWN

E_ZCL_ERR_ZBUFFER_FAIL

E_ZCL_ERR_ZTRANSMIT_FAIL

eCLD_PPGetPowerProfilePriceSend

```
teZCL_Status eCLD_PPGetPowerProfilePriceSend(
    uint8 u8SourceEndPointId,
    uint8 u8DestinationEndPointId,
    tsZCL_Address *psDestinationAddress,
    uint8 *pu8TransactionSequenceNumber,
    tsCLD_PP_PowerProfileReqPayload *psPayload);
```

Description

This function can be used on the cluster server to send a Get Power Profile Price Request to a cluster client, in order to request the cost of executing the schedule of a particular power profile. Use of this function must be enabled in the cluster compile-time options, as described in Section 7.10.

The function is non-blocking and will return immediately. On successfully receiving a Get Power Profile Price Response from the client, an E_CLD_PP_CMD_GET_POWER_PROFILE_PRICE_RSP event will be generated on the server, containing the requested price information in a `tsCLD_PP_GetPowerProfilePriceRspPayload` structure (see Section 7.9.9). For full details of handling a Get Power Profile Price Request, refer to Section 7.4.6.1.

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

Parameters

u8SourceEndPointId	Number of local endpoint on which cluster server resides
u8DestinationEndPointId	Number of remote endpoint on which cluster client resides
psDestinationAddress	Pointer to a structure containing the destination address of the client node
pu8TransactionSequenceNumber	Pointer to a location to receive the Transaction Sequence Number (TSN) of the message
psPayload	Pointer to structure containing the payload for the request (see Section 7.9.5), including the identifier of the relevant power profile

Returns

E_ZCL_SUCCESS

E_ZCL_FAIL

E_ZCL_ERR_EP_UNKNOWN

E_ZCL_ERR_ZBUFFER_FAIL

E_ZCL_ERR_ZTRANSMIT_FAIL

eCLD_PPGetPowerProfilePriceExtendedSend

```
teZCL_Status eCLD_PPGetPowerProfilePriceExtendedSend(
    uint8 u8SourceEndPointId,
    uint8 u8DestinationEndPointId,
    tsZCL_Address *psDestinationAddress,
    uint8 *pu8TransactionSequenceNumber,
    tsCLD_PP_GetPowerProfilePriceExtendedPayload
                                              *psPayload);
```

Description

This function can be used on the cluster server to send a Get Power Profile Price Extended Request to a cluster client, in order to request specific cost information about a power profile supported by the server. The cost of executing a power profile can be requested with either scheduled energy phases or contiguous energy phases (no gaps between them). Use of this function must be enabled in the cluster compile-time options, as described in Section 7.10.

The function is non-blocking and will return immediately. On successfully receiving a Get Power Profile Price Extended Response from the client, an E_CLD_PP_CMD_GET_POWER_PROFILE_PRICE_EXTENDED_RSP event will be generated on the server, containing the requested price information in a `tsCLD_PP_GetPowerProfilePriceRspPayload` structure (see Section 7.9.9). For full details of handling a Get Power Profile Price Extended Request, refer to Section 7.4.6.1.

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

Parameters

u8SourceEndPointId	Number of local endpoint on which cluster server resides
u8DestinationEndPointId	Number of remote endpoint on which cluster client resides
psDestinationAddress	Pointer to a structure containing the destination address of the client node
pu8TransactionSequenceNumber	Pointer to a location to receive the Transaction Sequence Number (TSN) of the message
psPayload	Pointer to structure containing the payload for the request (see Section 7.9.8), including the type of price information required

Returns

E_ZCL_SUCCESS
E_ZCL_FAIL
E_ZCL_ERR_EP_UNKNOWN
E_ZCL_ERR_ZBUFFER_FAIL
E_ZCL_ERR_ZTRANSMIT_FAIL

eCLD_PPGetOverallSchedulePriceSend

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

Description

This function can be used on the cluster server to send a Get Overall Schedule Price Request to a cluster client, in order to obtain the overall cost of all the power profiles that will be executed over the next 24 hours. Use of this function must be enabled in the cluster compile-time options, as described in Section 7.10.

The function is non-blocking and will return immediately. On successfully receiving a Get Overall Schedule Price Response from the client, an E_CLD_PP_CMD_GET_OVERALL_SCHEDULE_PRICE_RSP event will be generated on the server, containing the required price information in a `tsCLD_PP_GetOverallSchedulePriceRspPayload` structure (see Section 7.9.10). For full details of handling a Get Overall Schedule Price Request, refer to Section 7.4.6.2.

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

Parameters

u8SourceEndPointId	Number of local endpoint on which cluster server resides
u8DestinationEndPointId	Number of remote endpoint on which cluster client resides
psDestinationAddress	Pointer to a structure containing the destination address of the client node
pu8TransactionSequenceNumber	Pointer to a location to receive the Transaction Sequence Number (TSN) of the message

Returns

E_ZCL_SUCCESS

E_ZCL_FAIL

E_ZCL_ERR_EP_UNKNOWN

E_ZCL_ERR_ZBUFFER_FAIL

E_ZCL_ERR_ZTRANSMIT_FAIL

7.6.3 Client Functions

The following Power Profile cluster functions are provided in the HA API and can be used on a cluster client only:

Function	Page
eCLD_PPPowerProfileRequestSend	134
eCLD_PPEnergyPhasesScheduleNotificationSend	135
eCLD_PPPowerProfileStateReqSend	137
eCLD_PPEnergyPhasesScheduleStateReqSend	138
eCLD_PPPowerProfileScheduleConstraintsReqSend	139

eCLD_PPPowerProfileRequestSend

teZCL_Status eCLD_PPPowerProfileRequestSend(
uint8 *u8SourceEndPointId*,
uint8 *u8DestinationEndPointId*,
**tsZCL_Address ****psDestinationAddress*,
**uint8 ****pu8TransactionSequenceNumber*,
tsCLD_PP_PowerProfileReqPayload ****psPayload*);**

Description

This function can be used on a cluster client to send a Power Profile Request to the cluster server, in order to obtain one or more power profiles from the server. The function can be used to request a specific power profile (specified using its identifier) or all the power profiles supported by the server (specified using an identifier of zero).

The function is non-blocking and will return immediately. On receiving the server's response, an E_CLD_PP_CMD_POWER_PROFILE_RSP event will be generated on the client, containing a power profile in a `tsCLD_PP_PowerProfilePayload` structure (see Section 7.9.4). If all power profiles on the server have been requested, a response will be received for each profile separately and, therefore, the above event will be generated for each profile reported.

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

Parameters

u8SourceEndPointId	Number of local endpoint on which cluster client resides
u8DestinationEndPointId	Number of remote endpoint on which cluster server resides
psDestinationAddress	Pointer to a structure containing the destination address of the server node
pu8TransactionSequenceNumber	Pointer to a location to receive the Transaction Sequence Number (TSN) of the message
psPayload	Pointer to structure containing the payload for the request (see Section 7.9.3), including the identifier of the required power profile

Returns

E_ZCL_SUCCESS
E_ZCL_FAIL
E_ZCL_ERR_EP_UNKNOWN
E_ZCL_ERR_ZBUFFER_FAIL
E_ZCL_ERR_ZTRANSMIT_FAIL

eCLD_PPEnergyPhasesScheduleNotificationSend

```
teZCL_Status
eCLD_PPEnergyPhasesScheduleNotificationSend(
    uint8 u8SourceEndPointId,
    uint8 u8DestinationEndPointId,
    tsZCL_Address *psDestinationAddress,
    uint8 *pu8TransactionSequenceNumber,
    tsCLD_PP_EnergyPhasesSchedulePayload
                                    *psPayload);
```

Description

This function can be used on a cluster client to send an Energy Phases Schedule Notification to the cluster server, in order to start the schedule of energy phases of a power profile on the server. The function is used to send an unsolicited command and should only be called if the server allows itself to be remotely controlled. The command payload specifies the identifiers of the required energy phases and includes the relative start-times of the phases (see Section 7.9.12).

On receiving the notification on the server, the event E_CLD_PP_CMD_ENERGY_PHASE_SCHEDULE_NOTIFICATION will be generated, containing the sent energy phase schedule information in a `tsCLD_PP_EnergyPhasesSchedulePayload` structure (see Section 7.9.6). The subsequent handling of this notification is detailed in Section 7.4.4.2.

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

Parameters

u8SourceEndPointId	Number of local endpoint on which cluster client resides
u8DestinationEndPointId	Number of remote endpoint on which cluster server resides
psDestinationAddress	Pointer to a structure containing the destination address of the server node
pu8TransactionSequenceNumber	Pointer to a location to receive the Transaction Sequence Number (TSN) of the message
psPayload	Pointer to structure containing the payload for the command (see Section 7.9.6), including the scheduled energy phases and start-times

Returns

E_ZCL_SUCCESS

E_ZCL_FAIL

E_ZCL_ERR_EP_UNKNOWN

E_ZCL_ERR_ZBUFFER_FAIL

E_ZCL_ERR_ZTRANSMIT_FAIL

eCLD_PPPowerProfileStateReqSend

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

Description

This function can be used on a cluster client to send a Power Profile State Request to the cluster server, in order to obtain the identifier(s) of the power profile(s) currently supported on the server.

The function is non-blocking and will return immediately. On receiving the server's response, an E_CLD_PP_CMD_POWER_PROFILE_STATE_RSP event will be generated on the client, containing the required identifier(s). The response will contain the power profile records of all the supported power profiles on the server in a `tsCLD_PP_PowerProfileStatePayload` structure (see Section 7.9.5).

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

Parameters

u8SourceEndPointId	Number of local endpoint on which cluster client resides
u8DestinationEndPointId	Number of remote endpoint on which cluster server resides
psDestinationAddress	Pointer to a structure containing the destination address of the server node
pu8TransactionSequenceNumber	Pointer to a location to receive the Transaction Sequence Number (TSN) of the message

Returns

E_ZCL_SUCCESS

E_ZCL_FAIL

E_ZCL_ERR_EP_UNKNOWN

E_ZCL_ERR_ZBUFFER_FAIL

E_ZCL_ERR_ZTRANSMIT_FAIL

eCLD_PPEnergyPhasesScheduleStateReqSend

```
teZCL_Status
eCLD_PPEnergyPhasesScheduleStateReqSend(
    uint8 u8SourceEndPointId,
    uint8 u8DestinationEndPointId,
    tsZCL_Address *psDestinationAddress,
    uint8 *pu8TransactionSequenceNumber,
    tsCLD_PP_PowerProfileReqPayload *psPayload);
```

Description

This function can be used on a cluster client to send an Energy Phases Schedule State Request to the cluster server, in order to obtain the schedule of energy phases for a particular power profile on the server. The obtained schedule can be used to re-align the schedule information on the client with the information on the server - for example, after a reset of the client device.

The function is non-blocking and will return immediately. On receiving the server's response, an E_CLD_PP_CMD_ENERGY_PHASES_SCHEDULE_STATE_RSP event will be generated on the client, containing the requested schedule information in a `tsCLD_PP_EnergyPhasesSchedulePayload` structure (see Section 7.9.6).

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

Parameters

u8SourceEndPointId	Number of local endpoint on which cluster client resides
u8DestinationEndPointId	Number of remote endpoint on which cluster server resides
psDestinationAddress	Pointer to a structure containing the destination address of the server node
pu8TransactionSequenceNumber	Pointer to a location to receive the Transaction Sequence Number (TSN) of the message
psPayload	Pointer to structure containing the payload for the request (see Section 7.9.3), including the identifier of the relevant power profile

Returns

E_ZCL_SUCCESS

E_ZCL_FAIL

E_ZCL_ERR_EP_UNKNOWN

E_ZCL_ERR_ZBUFFER_FAIL

E_ZCL_ERR_ZTRANSMIT_FAIL

eCLD_PPPowerProfileScheduleConstraintsReqSend

```
teZCL_Status
eCLD_PPPowerProfileScheduleConstraintsReqSend(
    uint8 u8SourceEndPointId,
    uint8 u8DestinationEndPointId,
    tsZCL_Address *psDestinationAddress,
    uint8 *pu8TransactionSequenceNumber,
    tsCLD_PP_PowerProfileReqPayload *psPayload);
```

Description

This function can be used on a cluster client to send a Power Profile Schedule Constraints Request to the cluster server, in order to obtain the schedule restrictions on a particular power profile on the server. The obtained constraints can subsequently be used in calculating a schedule of energy phases for the power profile (e.g. before calling **eCLD_PPEnergyPhasesScheduleNotificationSend()**).

The function is non-blocking and will return immediately. The server will send a response to this request and on receiving this response, an E_CLD_PP_CMD_GET_POWER_PROFILE_SCHEDULE_CONSTRAINTS_RSP event will be generated on the client, containing the requested schedule constraints in a `tsCLD_PP_PowerProfileScheduleConstraintsPayload` structure (see Section 7.9.7).

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

Parameters

u8SourceEndPointId	Number of local endpoint on which cluster client resides
u8DestinationEndPointId	Number of remote endpoint on which cluster server resides
psDestinationAddress	Pointer to a structure containing the destination address of the server node
pu8TransactionSequenceNumber	Pointer to a location to receive the Transaction Sequence Number (TSN) of the message
psPayload	Pointer to structure containing the payload for the request (see Section 7.9.3), including the identifier of the relevant power profile

Returns

E_ZCL_SUCCESS
E_ZCL_FAIL
E_ZCL_ERR_EP_UNKNOWN
E_ZCL_ERR_ZBUFFER_FAIL
E_ZCL_ERR_ZTRANSMIT_FAIL

7.7 Return Codes

The Power Profile cluster functions use the ZCL return codes defined in the *ZCL User Guide (JN-UG-3077)*.

7.8 Enumerations

7.8.1 'Attribute ID' Enumerations

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

```
typedef enum PACK
{
    E_CLD_PP_ATTR_ID_TOTAL_PROFILE_NUM = 0x0000,
    E_CLD_PP_ATTR_ID_MULTIPLE_SCHEDULE,
    E_CLD_PP_ATTR_ID_ENERGY_FORMATTING,
    E_CLD_PP_ATTR_ID_ENERGY_REMOTE,
    E_CLD_PP_ATTR_ID_SCHEDULE_MODE
} teCLD_PP_Cluster_AttrID;
```

7.8.2 'Power Profile State' Enumerations

The following enumerations represent the possible states of a power profile.

```
typedef enum PACK
{
    E_CLD_PP_STATE_IDLE         = 0x00,
    E_CLD_PP_STATE_PROGRAMMED,
    E_CLD_PP_STATE_RUNNING      = 0x02,
    E_CLD_PP_STATE_PAUSED,
    E_CLD_PP_STATE_WAITING_TO_START,
    E_CLD_PP_STATE_WAITING_PAUSED,
    E_CLD_PP_STATE_ENDED,
} teCLD_PP_PowerProfileState;
```

The above enumerations are described in the table below.

Enumeration	Description
E_CLD_PP_STATE_IDLE	Not all parameters of the power profile have yet been defined
E_CLD_PP_STATE_PROGRAMMED	In the programmed state, as all the parameters of the power profile have been defined but there is no schedule or a schedule exists but has not been started
E_CLD_PP_STATE_RUNNING	The power profile is active and an energy phase is running
E_CLD_PP_STATE_PAUSED	The power profile is active but the current energy phase is paused
E_CLD_PP_STATE_WAITING_TO_START	The power profile is between two energy phases - one has ended and the next one has not yet started. If the next energy phase is the first energy phase of the schedule, this state indicates that schedule has been started but the first energy has not yet started
E_CLD_PP_STATE_WAITING_PAUSED	The power profile has been paused while in the 'waiting to start' state (described above)
E_CLD_PP_STATE_ENDED	The power profile schedule has finished

Table 37: 'Power Profile State' Enumerations

7.8.3 'Server-Generated Command' Enumerations

The following enumerations represent the commands that can be generated by the cluster server.

```
typedef enum PACK
{
 E_CLD_PP_CMD_POWER_PROFILE_NOTIFICATION = 0x00,
 E_CLD_PP_CMD_POWER_PROFILE_RSP,
 E_CLD_PP_CMD_POWER_PROFILE_STATE_RSP,
 E_CLD_PP_CMD_GET_POWER_PROFILE_PRICE,
 E_CLD_PP_CMD_POWER_PROFILE_STATE_NOTIFICATION,
 E_CLD_PP_CMD_GET_OVERALL_SCHEDULE_PRICE,
 E_CLD_PP_CMD_ENERGY_PHASES_SCHEDULE_REQ,
 E_CLD_PP_CMD_ENERGY_PHASES_SCHEDULE_STATE_RSP,
 E_CLD_PP_CMD_ENERGY_PHASES_SCHEDULE_STATE_NOTIFICATION,
 E_CLD_PP_CMD_GET_POWER_PROFILE_SCHEDULE_CONSTRAINTS_NOTIFICATION,
 E_CLD_PP_CMD_GET_POWER_PROFILE_SCHEDULE_CONSTRAINTS_RSP,
 E_CLD_PP_CMD_GET_POWER_PROFILE_PRICE_EXTENDED
 } teCLD_PP_ServerGeneratedCommandID;
```

The above enumerations are used to indicate types of Power Profile cluster events and are described in Section 7.5.

7.8.4 ‘Server-Received Command’ Enumerations

The following enumerations represent the commands that can be received by the cluster server (and are therefore generated on the cluster client).

```
typedef enum PACK
{
  E_CLD_PP_CMD_POWER_PROFILE_REQ = 0x00,
  E_CLD_PP_CMD_POWER_PROFILE_STATE_REQ,
  E_CLD_PP_CMD_GET_POWER_PROFILE_PRICE_RSP,
  E_CLD_PP_CMD_GET_OVERALL_SCHEDULE_PRICE_RSP,
  E_CLD_PP_CMD_ENERGY_PHASES_SCHEDULE_NOTIFICATION,
  E_CLD_PP_CMD_ENERGY_PHASES_SCHEDULE_RSP,
  E_CLD_PP_CMD_POWER_PROFILE_SCHEDULE_CONSTRAINTS_REQ,
  E_CLD_PP_CMD_ENERGY_PHASES_SCHEDULE_STATE_REQ,
  E_CLD_PP_CMD_GET_POWER_PROFILE_PRICE_EXTENDED_RSP
 } teCLD_PP_ServerReceivedCommandID;
```

The above enumerations are used to indicate types of Power Profile cluster events and are described in Section 7.5.

7.9 Structures

7.9.1 tsCLD_PPCallBackMessage

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

```
typedef struct
{
    uint8          u8CommandId;
#ifdef PP_CLIENT
    bool           bIsInfoAvailable;
#endif
    union
    {
    tsCLD_PP_PowerProfileReqPayload *psPowerProfileReqPayload;
    tsCLD_PP_GetPowerProfilePriceExtendedPayload
                                *psGetPowerProfilePriceExtendedPayload;
    } uReqMessage;
    union
    {
    tsCLD_PP_GetPowerProfilePriceRspPayload *psGetPowerProfilePriceRspPayload;
    tsCLD_PP_GetOverallSchedulePriceRspPayload
                                        *psGetOverallSchedulePriceRspPayload;
    tsCLD_PP_EnergyPhasesSchedulePayload    *psEnergyPhasesSchedulePayload;
    tsCLD_PP_PowerProfileScheduleConstraintsPayload
                                     *psPowerProfileScheduleConstraintsPayload;
    tsCLD_PP_PowerProfilePayload            *psPowerProfilePayload;
    tsCLD_PP_PowerProfileStatePayload       *psPowerProfileStatePayload;
    }uRespMessage;
} tsCLD_PPCallBackMessage;
```

where:

- `u8CommandId` indicates the type of Power Profile command that has been received, one of:
  - E_CLD_PP_CMD_POWER_PROFILE_REQ
  - E_CLD_PP_CMD_ POWER_PROFILE_STATE_REQ
  - E_CLD_PP_CMD_GET_POWER_PROFILE_PRICE_RSP
  - E_CLD_PP_CMD_GET_POWER_PROFILE_PRICE_EXTENDED_RSP
  - E_CLD_PP_CMD_GET_OVERALL_SCHEDULE_PRICE_RSP
  - E_CLD_PP_CMD_ENERGY_PHASES_SCHEDULE_NOTIFICATION
  - E_CLD_PP_CMD_ENERGY_PHASES_SCHEDULE_RSP
  - E_CLD_PP_CMD_GET_POWER_PROFILE_SCHEDULE_CONSTRAINTS_REQ
  - E_CLD_PP_CMD_ENERGY_PHASES_SCHEDULE_STATE_REQ
  - E_CLD_PP_CMD_POWER_PROFILE_NOTIFICATION
  - E_CLD_PP_CMD_POWER_PROFILE_RSP

  - E_CLD_PP_CMD_POWER_PROFILE_STATE_RSP
  - E_CLD_PP_CMD_POWER_PROFILE_STATE_NOTIFICATION
  - E_CLD_PP_CMD_GET_POWER_PROFILE_PRICE
  - E_CLD_PP_CMD_GET_OVERALL_SCHEDULE_PRICE
  - E_CLD_PP_CMD_ENERGY_PHASES_SCHEDULE_REQ
  - E_CLD_PP_CMD_ENERGY_PHASES_SCHEDULE_STATE_RSP
  - E_CLD_PP_CMD_ENERGY_PHASES_SCHEDULE_STATE_NOTIFICATION
  - E_CLD_PP_CMD_SCHEDULE_CONSTRAINTS_NOTIFICATION
  - E_CLD_PP_CMD_GET_POWER_PROFILE_SCHEDULE_CONSTRAINTS_RSP
  - E_CLD_PP_CMD_GET_POWER_PROFILE_PRICE_EXTENDED
- `bIsInfoAvailable` is a client-only boolean field which indicates whether the appropriate type of information (to which the event relates) is held on the client: TRUE if the information type is held on the client, FALSE otherwise
- `uReqMessage` is a union containing the command payload for a request, as one of (depending on the value of `u8CommandId`):
  - `psPowerProfileReqPayload` is a pointer to the payload of a Power Profile Request, a Get Power Profile Schedule Constraints Request, an Energy Phases Schedule Request, an Energy Phases Schedule State Request or a Get Power Profile Price Request (see Section 7.9.5)
  - `psGetPowerProfilePriceExtendedPayload` is a pointer to the payload of a Get Power Profile Price Extended Request (see Section 7.9.8)
- `uRespMessage` is a union containing the command payload for a response or notification, as one of (depending on the value of `u8CommandId`):
  - `psGetPowerProfilePriceRspPayload` is a pointer to the payload of a Get Power Profile Price Response or a Get Power Profile Price Extended Response (see Section 7.9.9)
  - `psGetOverallSchedulePriceRspPayload` is a pointer to the payload of a Get Overall Schedule Price Response (see Section 7.9.10)
  - `psEnergyPhasesSchedulePayload` is a pointer to the payload of an Energy Phases Schedule Response, an Energy Phases Schedule State Response, an Energy Phases Schedule Notification or an Energy Phases Schedule State Notification (see Section 7.9.6)
  - `psPowerProfileScheduleConstraintsPayload` is a pointer to the payload of a Power Profile Schedule Constraints Response or a Power Profile Schedule Constraints Notification (see Section 7.9.10)
  - `psPowerProfilePayload` is a pointer to the payload of a Power Profile Response or a Power Profile Notification (see Section 7.9.4)
  - `psPowerProfileStatePayload` is a pointer to the payload of a Power Profile State Response or a Power Profile State Notification (see Section 7.9.5)

Note: The command payload for each command type is indicated in Table 35 and Table 36 in Section 7.5.

7.9.2 tsCLD_PPEntry

This structure contains the data for a power profile table entry.

```
typedef struct
{
   zuint8   u8PowerProfileId;
   zuint8   u8NumOfTransferredEnergyPhases;
   zuint8   u8NumOfScheduledEnergyPhases;
   zuint8   u8ActiveEnergyPhaseId;
   tsCLD_PP_EnergyPhaseDelay
            asEnergyPhaseDelay[CLD_PP_NUM_OF_ENERGY_PHASES];
   tsCLD_PP_EnergyPhaseInfo
            asEnergyPhaseInfo[CLD_PP_NUM_OF_ENERGY_PHASES];
   zbool    bPowerProfileRemoteControl;
   zenum8   u8PowerProfileState;
   zuint16  u16StartAfter;
   zuint16  u16StopBefore;
} tsCLD_PPEntry;
```

where:

- `u8PowerProfileId` is the identifier of the power profile in the range 1 to 255 (0 is reserved)
- `u8NumOfTransferredEnergyPhases` is the number of energy phases supported within the power profile
- `u8NumOfScheduledEnergyPhases` is the number of energy phases actually scheduled within the power profile (must be less than or equal to the value of `u8NumOfTransferredEnergyPhases`)
- `u8ActiveEnergyPhaseId` is the identifier of the energy phase that is currently active or will be active next (if currently between energy phases)
- `asEnergyPhaseDelay[]` is an array containing timing information on the scheduled energy phases, where each array element corresponds to one energy phase of the schedule (see Section 7.9.12)
- `asEnergyPhaseInfo[]` is an array containing various information on the supported energy phases, where each array element corresponds to one energy phase of the profile (see Section 7.9.11)

- `bPowerProfileRemoteControl` is a boolean indicating whether the power profile can be remotely controlled: TRUE if it can be remotely controlled, FALSE otherwise (this property directly affects the attribute `bEnergyRemote`)
- `u8PowerProfileState` is a value indicating the current state of the power profile (enumerations are provided - see Section 7.8.2)
- `u16StartAfter` is the minimum time-delay, in minutes, to be implemented between an instruction to start the power profile schedule and actually starting the schedule
- `u16StopBefore` is the maximum time-delay, in minutes, to be implemented between an instruction to stop the power profile schedule and actually stopping the schedule

7.9.3 tsCLD_PP_PowerProfileReqPayload

This structure contains the payload of various power profile requests.

```
typedef struct
{
   zuint8   u8PowerProfileId;
}tsCLD_PP_PowerProfileReqPayload;
```

where `u8PowerProfileId` is the identifier of the power profile of interest.

7.9.4 tsCLD_PP_PowerProfilePayload

This structure contains the payload of a Power Profile Response or of a Power Profile Notification, which reports the details of a power profile.

```
typedef struct
{
   zuint8                    u8TotalProfileNum;
   zuint8                    u8PowerProfileId;
   zuint8                    u8NumOfTransferredPhases;
   tsCLD_PP_EnergyPhaseInfo *psEnergyPhaseInfo;
}tsCLD_PP_PowerProfilePayload;
```

where:

- `u8TotalProfileNum` is the total number of power profiles supported on the originating device
- `u8PowerProfileId` is the identifier of the power profile being reported
- `u8NumOfTransferredPhases` is the number of energy phases supported within the power profile
- `psEnergyPhaseInfo` is a pointer to a structure or an array of structures (see Section 7.9.11) containing information on the supported energy phases, where each array element corresponds to one energy phase of the profile

7.9.5 tsCLD_PP_PowerProfileStatePayload

This structure contains the payload of a Power Profile State Response or of a Power Profile State Notification.

```
typedef struct
{
    zuint8                      u8PowerProfileCount;
    tsCLD_PP_PowerProfileRecord *psPowerProfileRecord;
}tsCLD_PP_PowerProfileStatePayload;
```

where:

- `u8PowerProfileCount` is the number of power profiles in the payload
- `psPowerProfileRecord` is a pointer to one or more power profile records (see Section 7.9.13):
    - For a Power Profile State Notification, it is a pointer to the power profile record of the currently active power profile on the server
    - For a Power Profile State Response, it is a pointer to an array of power profile records for all the supported power profiles on the server

7.9.6 tsCLD_PP_EnergyPhasesSchedulePayload

This structure contains the payload of an Energy Phases Schedule Response, of an Energy Phases Schedule State Response or of an Energy Phases Schedule State Notification.

```
typedef struct
{
    zuint8                     u8PowerProfileId;
    zuint8                     u8NumOfScheduledPhases;
    tsCLD_PP_EnergyPhaseDelay *psEnergyPhaseDelay;
}tsCLD_PP_EnergyPhasesSchedulePayload;
```

where:

- `u8PowerProfileId` is the identifier of the power profile being reported
- `u8NumOfScheduledPhases` is the number of energy phases within the power profile schedule
- `psEnergyPhaseDelay` is a pointer to an array containing timing information on the scheduled energy phases, where each array element corresponds to one energy phase of the schedule (see Section 7.9.12)

7.9.7 tsCLD_PP_PowerProfileScheduleConstraintsPayload

This structure contains the payload of a Power Profile Schedule Constraints Response or of a Power Profile Schedule Constraints Notification, which reports the schedule restrictions on a particular power profile.

```
typedef struct
{
   zuint8   u8PowerProfileId;
   zuint16  u16StartAfter;
   zuint16  u16StopBefore;
} tsCLD_PP_PowerProfileScheduleConstraintsPayload;
```

where:

- `u8PowerProfileId` is the identifier of the power profile being reported
- `u16StartAfter` is the minimum time-delay, in minutes, to be implemented between an instruction to start the power profile schedule and actually starting the schedule
- `u16StopBefore` is the maximum time-delay, in minutes, to be implemented between an instruction to stop the power profile schedule and actually stopping the schedule

7.9.8 tsCLD_PP_GetPowerProfilePriceExtendedPayload

This structure contains the payload of a Get Power Profile Price Extended Request, which requests certain price information relating to a particular power profile.

```
typedef struct
{
    zbmap8    u8Options;
    zuint8    u8PowerProfileId;
    zuint16   u16PowerProfileStartTime;
}tsCLD_PP_GetPowerProfilePriceExtendedPayload;
```

where:

- `u8Options` is a bitmap indicating the type of request:
  - If bit 0 is set to ‘1’ then the `u16PowerProfileStartTime` field is used, otherwise it is ignored
  - If bit 1 is set to ‘0’ then an estimated price is required for contiguous energy phases (with no gaps between them); if bit 1 is set ‘1’ then an estimated price is required for the energy phases as scheduled (with any scheduled gaps between them)
- `u8PowerProfileId` is the identifier of the power profile

- `u16PowerProfileStartTime` is an optional value (see `u8Options` above) which indicates the required start-time for execution of the power profile, in minutes, as measured from the current time

7.9.9 tsCLD_PP_GetPowerProfilePriceRspPayload

This structure contains the payload of a Get Power Profile Price Response, which is returned in reply to a Get Power Profile Price Request and a Get Power Profile Price Extended Request.

```
typedef struct
{
     zuint8    u8PowerProfileId;
     zuint16   u16Currency;
     zuint32   u32Price;
     zuint8    u8PriceTrailingDigits;
}tsCLD_PP_GetPowerProfilePriceRspPayload;
```

where:

- `u8PowerProfileId` is the identifier of the power profile
- `u16Currency` is a value representing the currency in which the price is quoted
- `u32Price` is the price as an integer value (without a decimal point)
- `u8PriceTrailingDigits` specifies the position of the decimal point in the price `u32Price`, by indicating the number of digits after the decimal point

7.9.10 tsCLD_PP_GetOverallSchedulePriceRspPayload

This structure contains the payload of a Energy Phases Schedule Response, which contains the overall cost of all the power profiles that will be executed over the next 24 hours.

```
typedef struct
{
     zuint16   u16Currency;
     zuint32   u32Price;
     zuint8    u8PriceTrailingDigits;
}tsCLD_PP_GetOverallSchedulePriceRspPayload;
```

where:

- `u16Currency` is a value representing the currency in which the price is quoted
- `u32Price` represents the price as an integer value (without a decimal point)
- `u8PriceTrailingDigits` specifies the position of the decimal point in the price `u32Price`, by indicating the number of digits after the decimal point

7.9.11 tsCLD_PP_EnergyPhaseInfo

This structure contains various pieces of information about a specific energy phase of a power profile.

```
typedef struct
{
    zuint8    u8EnergyPhaseId;
    zuint8    u8MacroPhaseId;
    zuint16   u16ExpectedDuration;
    zuint16   u16PeakPower;
    zuint16   u16Energy;
    zuint16   u16MaxActivationDelay;
}tsCLD_PP_EnergyPhaseInfo;
```

where:

- `u8EnergyPhaseId` is the identifier of the energy phase
- `u8MacroPhaseId` is a value that may be used to obtain a name/label for the energy phase for display purposes - for example, it may be the index of an entry in a table of ASCII strings
- `u16ExpectedDuration` is the expected duration of the energy phase, in minutes
- `u16PeakPower` is the estimated peak power of the energy phase, in Watts
- `u16Energy` is the estimated total energy consumption, in Joules, during the energy phase (≤ `u16PeakPower` x `u16ExpectedDuration` x `60`)
- `u16MaxActivationDelay` is the maximum delay, in minutes, between the end of the previous energy phase and the start of this energy phase - special values are as follows: 0x0000 if no delay possible, 0xFFFF if first energy phase

Note: If `u16MaxActivationDelay` is non-zero, a delayed start-time for the energy phase can be set through the structure `tsCLD_PP_EnergyPhaseDelay` (see Section 7.9.12).

7.9.12 tsCLD_PP_EnergyPhaseDelay

This structure contains the start-time for a particular energy phase of a power profile.

```
typedef struct
{
     zuint8   u8EnergyPhaseId;
     zuint16  u16ScheduleTime;
}tsCLD_PP_EnergyPhaseDelay;
```

where:

- `u8EnergyPhaseId` is the identifier of the energy phase
- `u16ScheduleTime` is the start-time of the energy phase expressed as a delay, in minutes, from the end of the previous energy phase (for the first energy phase of a power profile schedule, this delay is measured from the start of the schedule)

Note: A delayed start-time for the energy phase can only be set through this structure if the field `u16MaxActivationDelay` of the structure `tsCLD_PP_EnergyPhaseInfo` for this energy phase is non-zero (see Section 7.9.11).

7.9.13 tsCLD_PP_PowerProfileRecord

This structure contains information about the current state of a power profile.

```
typedef struct
{
     zuint8   u8PowerProfileId;
     zuint8   u8EnergyPhaseId;
     zbool    bPowerProfileRemoteControl;
     zenum8   u8PowerProfileState;
} tsCLD_PP_PowerProfileRecord;
```

where:

- `u8PowerProfileId` is the identifier of the power profile
- `u8EnergyPhaseId` is the identifier of the currently running energy phase or, if currently between energy phases, the next energy phase to be run
- `bPowerProfileRemoteControl` is a boolean indicating whether the power profile can be remotely controlled (from a client device): TRUE if it can be remotely controlled, FALSE otherwise
- `u8PowerProfileState` is an enumeration indicating the current state of the power profile (see Section 7.8.2)

7.9.14 tsCLD_PPCustomDataStructure

The Power Profile cluster requires extra storage space to be allocated to be used by internal functions. The structure definition for this storage is shown below:

```
typedef struct
{
#ifdef (CLD_PP) && (PP_SERVER)
    bool                          bOverrideRunning;
    uint8                         u8ActSchPhaseIndex;
    tsCLD_PPEntry asPowerProfileEntry[CLD_PP_NUM_OF_POWER_PROFILES];
#endif
    tsZCL_ReceiveEventAddress     sReceiveEventAddress;
    tsZCL_CallBackEvent           sCustomCallBackEvent;
    tsCLD_PPCallBackMessage       sCallBackMessage;
} tsCLD_PPCustomDataStructure;
```

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

7.10 Compile-Time Options

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

To enable the Power Profile cluster in the code to be built, it is necessary to add the following line to the file:

```
#define CLD_PP
```

In addition, to enable the cluster as a client or server, it is also necessary to add one of the following lines to the same file:

```
#define PP_SERVER
#define PP_CLIENT
```

The following options can also be configured at compile-time in the **zcl_options.h** file.

Enable ‘Get Power Profile Price’ Command

The optional ‘Get Power Profile Price’ command can be enabled by adding:

```
#define CLD_PP_CMD_GET_POWER_PROFILE_PRICE
```

Enable ‘Get Power Profile Price Extended’ Command

The optional ‘Get Power Profile Price Extended’ command can be enabled by adding:

```
#define CLD_PP_CMD_GET_POWER_PROFILE_PRICE_EXTENDED
```

Enable ‘Get Overall Schedule Price’ Command

The optional ‘Get Overall Schedule Price’ command can be enabled by adding:

```
#define CLD_PP_CMD_GET_OVERALL_SCHEDULE_PRICE
```

Number of Power Profiles

The number of power profiles on the local device can be defined as n by adding:

```
#define CLD_PP_NUM_OF_PROFILES n
```

In the current HA release, this value is fixed at 1 (only one power profile is allowed on a device).

Maximum Number of Energy Phases

The maximum number of energy phases in a power profile can be defined as n by adding:

```
#define CLD_PP_NUM_OF_ENERGY_PHASES n
```

By default, this value is 3.

8. Appliance Control Cluster

This chapter outlines the Appliance Control cluster which is defined in the ZigBee Home Automation profile, and provides an interface for remotely controlling appliances in the home.

The Appliance Control cluster has a Cluster ID of 0x001B.

8.1 Overview

The Appliance Control cluster provides an interface for the remote control and programming of home appliances (e.g. a washing machine) by sending basic operational commands such as start, pause, stop.

The cluster is enabled by defining CLD_APPLIANCE_CONTROL in the **zcl_options.h** file - see Section 3.5.1. Further compile-time options for the Appliance Control cluster are detailed in Section 8.9.

All attributes of the Appliance Control cluster are in the 'Appliance Functions' attribute set.

8.2 Cluster Structure and Attributes

The Appliance Control cluster is contained in the following `tsCLD_ApplianceControl` structure:

```
typedef struct
{
    zuint16 u16StartTime;
    zuint16 u16FinishTime;

#ifdef CLD_APPLIANCE_CONTROL_REMAINING_TIME
    zuint16 u16RemainingTime;
#endif

} tsCLD_ApplianceControl;
```

where:

- `u16StartTime` is a bitmap representing the start-time of a 'running' cycle of the appliance, as follows:

Bits	Description
0-5	Minutes part of the start-time, in the range 0 to 59 (may be absolute or relative time - see below)
6-7	Type of time encoding: • 0x0: Relative time - start-time is a delay from the time that the attribute was set • 0x1: Absolute time - start-time is an actual time of the 24-hour clock • 0x2-0x3: Reserved The defaults are absolute time for ovens and relative time for other appliances.
8-15	Hours part of the start-time: • in the range 0 to 255, if relative time selected • in the range 0 to 23, if absolute time selected

- `u16FinishTime` is a bitmap representing the stop-time of a 'running' cycle of the appliance, as follows:

Bits	Description
0-5	Minutes part of the stop-time, in the range 0 to 59 (may be absolute or relative time - see below)
6-7	Type of time encoding: • 0x0: Relative time - stop-time is a delay from the time that the attribute was set • 0x1: Absolute time - stop-time is an actual time of the 24-hour clock • 0x2-0x3: Reserved The defaults are absolute time for ovens and relative time for other appliances.
8-15	Hours part of the stop-time: • in the range 0 to 255, if relative time selected • in the range 0 to 23, if absolute time selected

- `u16RemainingTime` is an optional attribute indicating the time, in minutes, remaining in the current 'running' cycle of the appliance (time until the end of the cycle) - this attribute is constantly updated during the running cycle and is zero when the appliance is not running

8.3 Sending Commands

The Appliance Control cluster server resides on the appliance to be controlled (e.g. a washing machine) and the cluster client resides on the controlling device (normally a remote control unit).

The commands from the client to the server can be of two types:

- ‘Execution’ commands, requesting appliance operations
- ‘Status’ commands, requesting appliance status information

In addition, status notification messages can be sent unsolicited from the server to the client.

Sending the above messages is described in the sub-sections below.

8.3.1 Execution Commands from Client to Server

An ‘execution’ command can be sent from the client to request that an operation is performed on the appliance (server) - the request is sent in an ‘Execution of Command’ message. The application on the client can send this message by calling the function **eCLD_ACExecutionOfCommandSend()**.

The possible operations depend on the target appliance but the following operations are available to be specified in the payload of the message (described in Section 8.8.2):

- Start appliance cycle
- Stop appliance cycle
- Pause appliance cycle
- Start superfreezing cycle
- Stop superfreezing cycle
- Start supercooling cycle
- Stop supercooling cycle
- Disable gas
- Enable gas

In the start and stop commands, the start-time and end-time can be specified. The commands are fully detailed in the British Standards document BS EN 50523.

The application on the server (appliance) will be notified of the received command by an E_CLD_APPLIANCE_CONTROL_CMD_EXECUTION_OF_COMMAND event (Appliance Control events are described in Section 8.4). The required command is specified in the payload of the message, which is contained in the above event. The application must then perform the requested command (if possible).

8.3.2 Status Commands from Client to Server

The application on the cluster client can request the current status of the appliance by sending a 'Signal State' message to the cluster server on the appliance. This message can be sent by calling the function **eCLD_ACSignalStateSend()**. This function returns immediately and the requested status information is later returned in an E_CLD_APPLIANCE_CONTROL_CMD_SIGNAL_STATE_RESPONSE event, which is generated when a response arrives from the server (Appliance Control events are described in Section 8.4).

> **Note:** The cluster server handles the 'Signal State' message automatically and returns the requested status information in a 'Signal State Response' message to the client.

The appliance status information from the message payload is contained in the above event - for details of this payload and the status information, refer to Section 8.8.3.

8.3.3 Status Notifications from Server to Client

The cluster server on the appliance can send unsolicited status notifications to the client in 'Signal State Notification' messages. A message of this kind can be sent by the application on the server by calling either of the following functions:

- **eCLD_ACSignalStateNotificationSend()**
- **eCLD_ACSignalStateResponseORSignalStateNotificationSend()**

> **Note:** The latter function is also used internally by the cluster server to send a 'Signal State Response' message - see Section 8.3.2.

The appliance status information from the 'Signal State Notification' message is reported to the application on the cluster client through the event E_CLD_APPLIANCE_CONTROL_CMD_SIGNAL_STATE_NOTIFICATION, which is generated when the notification arrives from the server (Appliance Control events are described in Section 8.4). The appliance status information from the message payload is contained in the above event - for details of this payload and the status information, refer to Section 8.8.3.

8.4 Appliance Control Events

The Appliance Control cluster has its own events that are handled through the callback mechanism outlined in Section 4.7 (and fully detailed in the *ZCL User Guide (JN-UG-3077)*). The cluster contains its own event handler. If a device uses this cluster then application-specific Appliance Control event handling must be included in the user-defined callback function for the associated endpoint, where this callback function is registered through the relevant endpoint registration function. This callback function will then be invoked when an Appliance Control event occurs and needs the attention of the application.

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

```
typedef struct
{
    uint8   u8CommandId;
    bool   *pbApplianceStatusTwoPresent;
    union
    {
        tsCLD_AC_ExecutionOfCommandPayload *psExecutionOfCommandPayload;
        tsCLD_AC_SignalStateResponseORSignalStateNotificationPayload
                                              *psSignalStateResponseAndNotificationPayload;
    } uMessage;
} tsCLD_ApplianceControlCallBackMessage;
```

When an Appliance Control event occurs, one of four command types could have been received. The relevant command type is specified through the `u8CommandId` field of the `tsSM_CallBackMessage` structure. The possible command types are detailed the tables below for events generated on a server and a client.

u8CommandId Enumeration	Description
E_CLD_APPLIANCE_CONTROL_ CMD_EXECUTION_OF_COMMAND	An 'Execution of Command' message has been received by the server (appliance), requesting an operation on the appliance
E_CLD_APPLIANCE_CONTROL_ CMD_SIGNAL_STATE	A 'Signal State' message has been received by the server (appliance), requesting the status of the appliance

Table 38: Appliance Control Command Types (Events on Server)

u8CommandId Enumeration	Description
E_CLD_APPLIANCE_CONTROL_CMD_SIGNAL_STATE_RESPONSE	A response to a 'Signal State' message has been received by the client, containing the requested appliance status
E_CLD_APPLIANCE_CONTROL_CMD_SIGNAL_STATE_NOTIFICATION	A 'Signal State' notification message has been received by the client, containing unsolicited status information

Table 39: Appliance Control Command Types (Events on Client)

8.5 Functions

The following Appliance Control cluster functions are provided in the HA API:

Function	Page
eCLD_ApplianceControlCreateApplianceControl	161
eCLD_ACExecutionOfCommandSend	163
eCLD_ACSignalStateSend	165
eCLD_ACSignalStateResponseORSignalStateNotificationSend	166
eCLD_ACSignalStateNotificationSend	168
eCLD_ACChangeAttributeTime	170

eCLD_ApplianceControlCreateApplianceControl

teZCL_Status
eCLD_ApplianceControlCreateApplianceControl(
tsZCL_ClusterInstance **psClusterInstance*,
bool_t *bIsServer*,
tsZCL_ClusterDefinition **psClusterDefinition*,
void **pvEndPointSharedStructPtr*,
uint8 **pu8AttributeControlBits*,
tsCLD_ApplianceControlCustomDataStructure
psCustomDataStructure*);**

Description

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

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

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

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

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

The array declaration should be as follows:

```
uint8 au8AppApplianceControlClusterAttributeControlBits[
CLD_APPLIANCE_CONTROL_MAX_NUMBER_OF_ATTRIBUTE];
```

The function will initialise the array elements to zero.

Parameters

psClusterInstance	Pointer to structure containing information about the cluster instance to be created (see the *ZCL User Guide (JN-UG-3077)*). This structure will be updated by the function by initialising individual structure fields.
bIsServer	Type of cluster instance (server or client) to be created: TRUE - server FALSE - client
psClusterDefinition	Pointer to structure indicating the type of cluster to be created (see the *ZCL User Guide (JN-UG-3077)*). In this case, this structure must contain the details of the Appliance Control cluster. This parameter can refer to a pre-filled structure called `sCLD_ApplianceControl` which is provided in the **ApplianceControl.h** file.
pvEndPointSharedStructPtr	Pointer to the shared structure used for attribute storage. This parameter should be the address of the structure of type `tsCLD_ApplianceControl` which defines the attributes of Appliance Control cluster. The function will initialise the attributes with default values.
pu8AttributeControlBits	Pointer to an array of **uint8** values, with one element for each attribute in the cluster (see above).
psCustomDataStructure	Pointer to a structure containing the storage for internal functions of the cluster (see Section 8.8.4)

Returns

E_ZCL_SUCCESS
E_ZCL_FAIL
E_ZCL_ERR_PARAMETER_NULL
E_ZCL_ERR_INVALID_VALUE

eCLD_ACExecutionOfCommandSend

teZCL_Status eCLD_ACExecutionOfCommandSend(
uint8 *u8SourceEndPointId*,
uint8 *u8DestinationEndPointId*,
tsZCL_Address **psDestinationAddress*,
uint8 **pu8TransactionSequenceNumber*,
tsCLD_AC_ExecutionOfCommandPayload **psPayload*);

Description

This function can be used on an Appliance Control cluster client to send an 'Execution of Command' message to a cluster server (appliance), where this message may specify one of the following control commands:

- Start appliance cycle
- Stop appliance cycle
- Pause appliance cycle
- Start superfreezing cycle
- Stop superfreezing cycle
- Start supercooling cycle
- Stop supercooling cycle
- Disable gas
- Enable gas

The required command is specified in the payload of the message (a pointer to this payload must be provided). The commands are fully detailed in the British Standards document BS EN 50523.

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

Parameter

u8SourceEndPointId	Number of the local endpoint through which to send the message. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values
u8DestinationEndPointId	Number of the endpoint on the remote node to which the message will be sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP
psDestinationAddress	Pointer to a structure holding the address of the node to which the request will be sent

pu8TransactionSequenceNumber	Pointer to a location to receive the Transaction Sequence Number (TSN) of the message
psPayload	Pointer to a structure containing the payload for the message (see Section 8.8.2).

Returns

E_ZCL_SUCCESS
E_ZCL_FAIL
E_ZCL_ERR_PARAMETER_NULL
E_ZCL_ERR_INVALID_VALUE

eCLD_ACSignalStateSend

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

Description

This function can be used on an Appliance Control cluster client to send a 'Signal State' message to a cluster server (appliance), which requests the status of the appliance. The function returns immediately and the requested status information is later returned in the following event, which is generated when a response is received from the server:

E_CLD_APPLIANCE_CONTROL_CMD_SIGNAL_STATE_RESPONSE

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

Parameters

u8SourceEndPointId	Number of the local endpoint through which to send the message. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values
u8DestinationEndPointId	Number of the endpoint on the remote node to which the message will be sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP
psDestinationAddress	Pointer to a structure holding the address of the node to which the message will be sent
pu8TransactionSequenceNumber	Pointer to a location to receive the Transaction Sequence Number (TSN) of the message

Returns

E_ZCL_SUCCESS
E_ZCL_FAIL
E_ZCL_ERR_PARAMETER_NULL
E_ZCL_ERR_INVALID_VALUE

eCLD_ACSignalStateResponseORSignalStateNotificationSend

```
teZCL_Status
eCLD_ACSignalStateResponseORSignalStateNotificationSend(
    uint8 u8SourceEndPointId,
    uint8 u8DestinationEndPointId,
    tsZCL_Address *psDestinationAddress,
    uint8 *pu8TransactionSequenceNumber,
    teCLD_ApplianceControl_ServerCommandId eCommandId,
    bool bApplianceStatusTwoPresent,
    tsCLD_AC_SignalStateResponseORSignalStateNotificationPayload
                                                    *psPayload);
```

Description

This function can be used on an Appliance Control cluster server to send a 'Signal State Response' message (in reply to a 'Signal State Request' message) or an unsolicited 'Signal State Notification' message to a cluster client.

The command to be sent must be specified as one of:

- E_CLD_APPLIANCE_CONTROL_CMD_SIGNAL_STATE_RESPONSE
- E_CLD_APPLIANCE_CONTROL_CMD_SIGNAL_STATE_NOTIFICATION

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

Parameters

u8SourceEndPointId	Number of the local endpoint through which to send the message. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values
u8DestinationEndPointId	Number of the endpoint on the remote node to which the message will be sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP
psDestinationAddress	Pointer to a structure holding the address of the node to which the message will be sent
pu8TransactionSequenceNumber	Pointer to a location to receive the Transaction Sequence Number (TSN) of the message
eCommandId	Enumeration indicating the command to be sent (see above and Section 8.7.3)

bApplianceStatusTwoPresent	Boolean indicating whether additional appliance status data is present in payload: TRUE - Present FALSE - Not present
psPayload	Pointer to structure containing payload for message (see above and Section 8.8.3)

Returns

E_ZCL_SUCCESS
E_ZCL_FAIL
E_ZCL_ERR_PARAMETER_NULL
E_ZCL_ERR_INVALID_VALUE

eCLD_ACSignalStateNotificationSend

teZCL_Status eCLD_ACSignalStateNotificationSend(
uint8 *u8SourceEndPointId*,
uint8 *u8DestinationEndPointId*,
tsZCL_Address ***psDestinationAddress*,
uint8 ***pu8TransactionSequenceNumber*,
bool *bAppliance StatusTwoPresent*,
tsCLD_AC_SignalStateResponseORSignalStateNotificationPayload
***psPayload*);**

Description

This function can be used on an Appliance Control cluster server to send an unsolicited ‘Signal State Notification’ message to a cluster client. The function is an alternative to **eCLD_ACSignalStateResponseORSignalStateNotificationSend()**.

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

Parameters

u8SourceEndPointId	Number of the local endpoint through which to send the message. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values
u8DestinationEndPointId	Number of the endpoint on the remote node to which the message will be sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP
psDestinationAddress	Pointer to a structure holding the address of the node to which the message will be sent
pu8TransactionSequenceNumber	Pointer to a location to receive the Transaction Sequence Number (TSN) of the message
bApplianceStatusTwoPresent	Boolean indicating whether additional appliance status data is present in payload: TRUE - Present FALSE - Not present
psPayload	Pointer to structure containing payload for message (see above and Section 8.8.3)

Returns

E_ZCL_SUCCESS
E_ZCL_FAIL
E_ZCL_ERR_PARAMETER_NULL
E_ZCL_ERR_INVALID_VALUE

eCLD_ACChangeAttributeTime

```
teZCL_Status eCLD_ACChangeAttributeTime(
        uint8 u8SourceEndPointId,
        teCLD_ApplianceControl_Cluster_AttrID eAttributeTimeId,
        uint16 u16TimeValue);
```

Description

This function can be used on an Appliance Control cluster server (appliance) to update the time attributes of the cluster (start time, finish time, remaining time). This is particularly useful if the host node has its own timer.

The target attribute must be specified using one of:

- E_CLD_APPLIANCE_CONTROL_ATTR_ID_START_TIME
- E_CLD_APPLIANCE_CONTROL_ATTR_ID_FINISH_TIME
- E_CLD_APPLIANCE_CONTROL_ATTR_ID_REMAINING_TIME

Parameters

u8SourceEndPointId	Number of the local endpoint through which to send the message. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values
eAttributeTimeId	Identifier of attribute to be updated (see above and Section 8.8.1)
u16TimeValue	UTC time to set

Returns

E_ZCL_SUCCESS
E_ZCL_FAIL
E_ZCL_ERR_PARAMETER_NULL
E_ZCL_ERR_INVALID_VALUE

8.6 Return Codes

The Appliance Control cluster functions use the ZCL return codes defined in the *ZCL User Guide (JN-UG-3077)*.

8.7 Enumerations

8.7.1 'Attribute ID' Enumerations

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

```
typedef enum PACK
{
    E_CLD_APPLIANCE_CONTROL_ATTR_ID_START_TIME = 0x0000,
    E_CLD_APPLIANCE_CONTROL_ATTR_ID_FINISH_TIME,
    E_CLD_APPLIANCE_CONTROL_ATTR_ID_REMAINING_TIME
} teCLD_ApplianceControl_Cluster_AttrID;
```

8.7.2 'Client Command ID' Enumerations

The following enumerations are used in commands issued on a cluster client.

```
typedef enum PACK
{
    E_CLD_APPLIANCE_CONTROL_CMD_EXECUTION_OF_COMMAND = 0x00,
    E_CLD_APPLIANCE_CONTROL_CMD_SIGNAL_STATE,
} teCLD_ApplianceControl_ClientCommandId;
```

The above enumerations are described in the table below.

Enumeration	Description
E_CLD_APPLIANCE_CONTROL_CMD_EXECUTION_OF_COMMAND	'Execution of Command' message
E_CLD_APPLIANCE_CONTROL_CMD_SIGNAL_STATE	'Signal State' message

Table 40: 'Client Command ID' Enumerations

8.7.3 'Server Command ID' Enumerations

The following enumerations are used in commands issued on a cluster server.

```
typedef enum PACK
{
    E_CLD_APPLIANCE_CONTROL_CMD_SIGNAL_STATE_RESPONSE = 0x00,
    E_CLD_APPLIANCE_CONTROL_CMD_SIGNAL_STATE_NOTIFICATION
} teCLD_ApplianceControl_ServerCommandId;
```

The above enumerations are described in the table below.

Enumeration	Description
E_CLD_APPLIANCE_CONTROL_CMD_SIGNAL_STATE_RESPONSE	A response to a 'Signal State' request
E_CLD_APPLIANCE_CONTROL_CMD_SIGNAL_STATE_NOTIFICATION	A 'Signal State' notification

Table 41: 'Server Command ID' Enumerations

8.8 Structures

8.8.1 tsCLD_ApplianceControlCallBackMessage

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

```
typedef struct
{
    uint8    u8CommandId;
    bool    *pbApplianceStatusTwoPresent;
    union
    {
        tsCLD_AC_ExecutionOfCommandPayload *psExecutionOfCommandPayload;
        tsCLD_AC_SignalStateResponseORSignalStateNotificationPayload
                                            *psSignalStateResponseAndNotificationPayload;
    } uMessage;
} tsCLD_ApplianceControlCallBackMessage;
```

where:

- u8CommandId indicates the type of Appliance Control command that has been received, one of:
  - E_CLD_APPLIANCE_CONTROL_CMD_EXECUTION_OF_COMMAND
  - E_CLD_APPLIANCE_CONTROL_CMD_SIGNAL_STATE
  - E_CLD_APPLIANCE_CONTROL_CMD_SIGNAL_STATE_RESPONSE
  - E_CLD_APPLIANCE_CONTROL_CMD_SIGNAL_STATE_NOTIFICATION
- pbApplianceStatusTwoPresent is a pointer to a boolean indicating whether a second set of non-standard or proprietary status data is available:
  - TRUE - additional status data available
  - FALSE - additional status data unavailable
- uMessage is a union containing the command payload as one of (depending on the value of u8CommandId):
  - psExecutionOfCommandPayload is a pointer to the payload of an ‘Execution of Command’ message (see Section 8.8.2)
  - psSignalStateResponseAndNotificationPayload is a pointer to the payload of a ‘Signal State’ response or notification message (see Section 8.8.3)

8.8.2 tsCLD_AC_ExecutionOfCommandPayload

This structure contains the payload for an “Execution of Command” message.

```
typedef struct
{
   zenum8  eExecutionCommandId;
} tsCLD_AC_ExecutionOfCommandPayload;
```

where `eExecutionCommandId` is a value representing the command to be executed - the commands are detailed in the British Standards document BS EN 50523.

8.8.3 tsCLD_AC_SignalStateResponseORSignalStateNotificationPayload

This structure contains the payload for a “Signal State” response or notification message.

```
typedef struct
{
   zenum8   eApplianceStatus;
   zuint8   u8RemoteEnableFlagAndDeviceStatus;
   zuint24  u24ApplianceStatusTwo;
} tsCLD_AC_SignalStateResponseORSignalStateNotificationPayload;
```

where:

- `eApplianceStatus` is a value indicating the reported appliance status (the relevant status values depend on the appliance):

Status Value	Description
0x00	Reserved
0x01	Appliance in off state
0x02	Appliance in stand-by
0x03	Appliance already programmed
0x04	Appliance already programmed and ready to start
0x05	Appliance is running
0x06	Appliance is in pause state
0x07	Appliance end programmed tasks
0x08	Appliance is in a failure state
0x09	Appliance programmed tasks have been interrupted
0x0A	Appliance in idle state
0x0B	Appliance rinse hold

Status Value	Description
0x0C	Appliance in service state
0x0D	Appliance in superfreezing state
0x0E	Appliance in supercooling state
0x0F	Appliance in superheating state
0x10-0x3F	Reserved
0x40-0x7F	Non-standardised
0x80-0xFF	Proprietary

- `u8RemoteEnableFlagAndDeviceStatus` is a bitmap value indicating the status of the relationship between the appliance and the remote control unit as well as the type of additional status information reported in `u24ApplianceStatusTwo`:

Bits	Field	Values/Description
0-3	Remote Enable Flags	Status of remote control link: • 0x0: Disabled • 0x1: Enabled remote and energy control • 0x2-0x06: Reserved • 0x7: Temporarily locked/disabled • 0x8-0xE: Reserved • 0xF: Enabled remote control
4-7	Device Status 2	Type of information in `u24ApplianceStatusTwo`: • 0x0: Proprietary • 0x1: Proprietary • 0x2: IRIS symptom code • 0x3-0xF: Reserved

- `u24ApplianceStatusTwo` is a value indicating non-standard or proprietary status information about the appliance. The type of status information represented by this value is indicated in the ‘Device Status 2’ field of `u8RemoteEnableFlagAndDeviceStatus`. In the case of an IRIS symptom code, the three bytes of this value represent a 3-digit code.

8.8.4 tsCLD_ApplianceControlCustomDataStructure

The Appliance Control cluster requires extra storage space to be allocated to be used by internal functions. The structure definition for this storage is shown below:

```
typedef struct
{
    tsZCL_ReceiveEventAddress                sReceiveEventAddress;
    tsZCL_CallBackEvent                      sCustomCallBackEvent;
    tsCLD_ApplianceControlCallBackMessage     sCallBackMessage;
} tsCLD_ApplianceControlCustomDataStructure;
```

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

8.9 Compile-Time Options

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

To enable the Appliance Control cluster in the code to be built, it is necessary to add the following line to the file:

```
#define CLD_APPLIANCE_CONTROL
```

In addition, to enable the cluster as a client or server, it is also necessary to add one of the following lines to the same file:

```
#define APPLIANCE_CONTROL_SERVER
#define APPLIANCE_CONTROL_CLIENT
```

The Appliance Control cluster has one optional attribute (see Section 8.2) which can be enabled using a macro that may be optionally specified at compile time by adding the following line to the **zcl_options.h** file:

```
#define CLD_APPLIANCE_CONTROL_REMAINING_TIME
```

9. Appliance Identification Cluster

This chapter outlines the Appliance Identification cluster which is defined in the ZigBee Home Automation profile, and provides an interface for obtaining and setting basic appliance information.

The Appliance Identification cluster has a Cluster ID of 0x0B00.

9.1 Overview

The Appliance Identification cluster provides an interface for obtaining and setting information about an appliance, such as product type and manufacturer.

The cluster is enabled by defining CLD_APPLIANCE_IDENTIFICATION in the **zcl_options.h** file - see Section 3.5.1. Further compile-time options for the Appliance Identification cluster are detailed in Section 9.6.

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

- Basic Appliance Identification
- Extended Appliance Identification

9.2 Cluster Structure and Attributes

The Appliance Identification cluster is contained in the following `tsCLD_Applianceidentification` structure:

```
typedef struct
{
    zbmap56                 u64BasicIdentification;

#ifdef CLD_APPLIANCE_IDENTIFICATION_ATTR_COMPANY_NAME
    tsZCL_CharacterString   sCompanyName;
    uint8                   au8CompanyName[16];
#endif

#ifdef CLD_APPLIANCE_IDENTIFICATION_ATTR_COMPANY_ID
    zuint16                 u16CompanyId;
#endif

#ifdef CLD_APPLIANCE_IDENTIFICATION_ATTR_BRAND_NAME
    tsZCL_CharacterString   sBrandName;
    uint8                   au8BrandName[16];
#endif
#ifdef CLD_APPLIANCE_IDENTIFICATION_ATTR_BRAND_ID
    zuint16             u16BrandId;
#endif

#ifdef CLD_APPLIANCE_IDENTIFICATION_ATTR_MODEL
    tsZCL_OctetString    sModel;
    uint8               au8Model[16];
#endif

#ifdef CLD_APPLIANCE_IDENTIFICATION_ATTR_PART_NUMBER
    tsZCL_OctetString    sPartNumber;
    uint8               au8PartNumber[16];
#endif

#ifdef CLD_APPLIANCE_IDENTIFICATION_ATTR_PRODUCT_REVISION
    tsZCL_OctetString    sProductRevision;
    uint8               au8ProductRevision[6];
#endif

#ifdef CLD_APPLIANCE_IDENTIFICATION_ATTR_SOFTWARE_REVISION
    tsZCL_OctetString    sSoftwareRevision;
    uint8               au8SoftwareRevision[6];
#endif

#ifdef CLD_APPLIANCE_IDENTIFICATION_ATTR_PRODUCT_TYPE_NAME
    tsZCL_OctetString    sProductTypeName;
    uint8               au8ProductTypeName[2];
#endif

#ifdef CLD_APPLIANCE_IDENTIFICATION_ATTR_PRODUCT_TYPE_ID
    zuint16             u16ProductTypeId;
#endif

#ifdef CLD_APPLIANCE_IDENTIFICATION_ATTR_CECED_SPEC_VERSION
    zuint8              u8CECEDSpecificationVersion;
#endif

} tsCLD_Applianceidentification;
```

where:

'Basic Appliance Identification' Attribute Set

- `u64BasicIdentification` is a mandatory attribute which is a 56-bit bitmap containing the following information about the appliance:

Bits	Information
0-15	Company (manufacturer) ID
16-31	Brand ID
32-47	Product Type ID, one of: • 0x0000: White Goods • 0x5601: Dishwasher • 0x5602: Tumble Dryer • 0x5603: Washer Dryer • 0x5604: Washing Machine • 0x5E03: Hob • 0x5E09: Induction Hob • 0x5E01: Oven • 0x5E06: Electrical Oven • 0x6601: Refrigerator/Freezer For enumerations, see Section 9.5.2.
48-55	Specification Version

'Extended Appliance Identification' Attribute Set

- The following optional pair of attributes are used to store human readable versions of the company (manufacturer) name:
  - `sCompanyName` is a `tsZCL_OctetString` structure which contains a character string representing the company name of up to 16 characters
  - `au8CompanyName[16]` is a byte-array which contains the character data bytes representing the company name
- `u16CompanyId` is an optional attribute which contains the company ID
- The following optional pair of attributes are used to store human readable versions of the brand name:
  - `sBrandName` is a `tsZCL_OctetString` structure which contains a character string representing the brand name of up to 16 characters
  - `au8BrandName[16]` is a byte-array which contains the character data bytes representing the brand name
- `u16BrandId` is an optional attribute which contains the brand ID
- The following optional pair of attributes are used to store human readable versions of the manufacturer-defined model name:
  - `sModel` is a `tsZCL_OctetString` structure which contains a character string representing the model name of up to 16 characters

    - `au8Model[16]` is a byte-array which contains the character data bytes representing the model name

- The following optional pair of attributes are used to store human readable versions of the manufacturer-defined part number/code:
    - `sPartNumber` is a `tsZCL_OctetString` structure which contains a character string representing the part number/code of up to 16 characters
    - `au8PartNumber[16]` is a byte-array which contains the character data bytes representing the part number/code
- The following optional pair of attributes are used to store human readable versions of the manufacturer-defined product revision number:
    - `sProductRevision` is a `tsZCL_OctetString` structure which contains a character string representing the product revision number of up to 6 characters
    - `au8ProductRevision[6]` is a byte-array which contains the character data bytes representing the product revision number
- The following optional pair of attributes are used to store human readable versions of the manufacturer-defined software revision number:
    - `sSoftwareRevision` is a `tsZCL_OctetString` structure which contains a character string representing the software revision number of up to 6 characters
    - `au8SoftwareRevision[6]` is a byte-array which contains the character data bytes representing the software revision number
- The following optional pair of attributes are used to store human readable versions of the 2-character product type name (e.g. “WM” for washing machine):
    - `sProductTypeName` is a `tsZCL_OctetString` structure which contains a character string representing the product type name of up to 2 characters
    - `au8ProductTypeName[2]` is a byte-array which contains the character data bytes representing the product type name
- `u16ProductTypeId` is an optional attribute containing the product type ID (from those listed above in the description of `u64BasicIdentification`)
- `u8CECEDSpecificationVersion` is an optional attribute which indicates the version of the CECED specification to which the appliance conforms, from the following:

Value	Specification
0x10	Compliant with v1.0, not certified
0x1A	Compliant with v1.0, certified
0xX0	Compliant with vX.0, not certified
0xXA	Compliant with vX.0, certified

9.3 Functions

The following Appliance Identification cluster function is provided in the HA API:

Function	Page
eCLD_ApplianceIdentificationCreateApplianceIdentification	182

Note: The attributes of this cluster can be accessed using the attribute read/write functions provided in the ZigBee Cluster Library and described in the *ZCL User Guide (JN-UG-3077)*.

eCLD_ApplianceIdentificationCreateApplianceIdentification

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

Description

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

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

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

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

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

The array declaration should be as follows:

```
uint8 au8AppApplianceIdentificationClusterAttributeControlBits[
CLD_APPLIANCE_IDENTIFICATION_MAX_NUMBER_OF_ATTRIBUTE];
```

The function will initialise the array elements to zero.

Parameters

psClusterInstance	Pointer to structure containing information about the cluster instance to be created (see the *ZCL User Guide (JN-UG-3077)*). This structure will be updated by the function by initialising individual structure fields.
bIsServer	Type of cluster instance (server or client) to be created: TRUE - server FALSE - client
psClusterDefinition	Pointer to structure indicating the type of cluster to be created (see the *ZCL User Guide (JN-UG-3077)*). In this case, this structure must contain the details of the Appliance Identification cluster. This parameter can refer to a pre-filled structure called `sCLD_ApplianceIdentification` which is provided in the **ApplianceIdentification.h** file.
pvEndPointSharedStructPtr	Pointer to the shared structure used for attribute storage. This parameter should be the address of the structure of the `tsCLD_ApplianceIdentification` type which defines the attributes of Appliance Identification cluster. The function will initialise the attributes with default values.
pu8AttributeControlBits	Pointer to an array of **uint8** values, with one element for each attribute in the cluster (see above).

Returns

E_ZCL_SUCCESS

E_ZCL_FAIL

E_ZCL_ERR_PARAMETER_NULL

E_ZCL_ERR_INVALID_VALUE

9.4 Return Codes

The Appliance Identification cluster function uses the ZCL return codes defined in the *ZCL User Guide (JN-UG-3077)*.

9.5 Enumerations

9.5.1 'Attribute ID' Enumerations

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

```
typedef enum PACK
{
    E_CLD_APPLIANCE_IDENTIFICATION_ATTR_ID_BASIC_IDENTIFICATION = 0x0000,
    E_CLD_APPLIANCE_IDENTIFICATION_ATTR_ID_COMPANY_NAME = 0x0010,
    E_CLD_APPLIANCE_IDENTIFICATION_ATTR_ID_COMPANY_ID,
    E_CLD_APPLIANCE_IDENTIFICATION_ATTR_ID_BRAND_NAME,
    E_CLD_APPLIANCE_IDENTIFICATION_ATTR_ID_BRAND_ID,
    E_CLD_APPLIANCE_IDENTIFICATION_ATTR_ID_MODEL,
    E_CLD_APPLIANCE_IDENTIFICATION_ATTR_ID_PART_NUMBER,
    E_CLD_APPLIANCE_IDENTIFICATION_ATTR_ID_PRODUCT_REVISION,
    E_CLD_APPLIANCE_IDENTIFICATION_ATTR_ID_SOFTWARE_REVISION,
    E_CLD_APPLIANCE_IDENTIFICATION_ATTR_ID_PRODUCT_TYPE_NAME,
    E_CLD_APPLIANCE_IDENTIFICATION_ATTR_ID_PRODUCT_TYPE_ID,
    E_CLD_APPLIANCE_IDENTIFICATION_ATTR_ID_CECED_SPEC_VERSION
} teCLD_ApplianceIdentification_Cluster_AttrID;
```

9.5.2 'Product Type ID' Enumerations

The following enumerations are used to represent the set of product type IDs.

```
typedef enum PACK
{
    E_CLD_AI_PT_ID_WHITE_GOODS              = 0x0000,
    E_CLD_AI_PT_ID_DISHWASHER               = 0x5601,
    E_CLD_AI_PT_ID_TUMBLE_DRYER,
    E_CLD_AI_PT_ID_WASHER_DRYER,
    E_CLD_AI_PT_ID_WASHING_MACHINE,
    E_CLD_AI_PT_ID_HOBS                     = 0x5E03,
    E_CLD_AI_PT_ID_INDUCTION_HOBS           = 0x5E09,
    E_CLD_AI_PT_ID_OVEN                     = 0x5E01,
    E_CLD_AI_PT_ID_ELECTRICAL_OVEN          = 0x5E06,
    E_CLD_AI_PT_ID_REFRIGERATOR_FREEZER     = 0x6601
} teCLD_ApplianceIdentification_ProductTypeId;
```

9.6 Compile-Time Options

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

To enable the Appliance Identification cluster in the code to be built, it is necessary to add the following line to the file:

```
#define CLD_APPLIANCE_IDENTIFICATION
```

In addition, to enable the cluster as a client or server, it is also necessary to add one of the following lines to the same file:

```
#define APPLIANCE_IDENTIFICATION_SERVER
#define APPLIANCE_IDENTIFICATION_CLIENT
```

Optional Attributes

The optional attributes for the Appliance Identification cluster (see Section 9.2) are enabled by defining:

- CLD_APPLIANCE_IDENTIFICATION_ATTR_COMPANY_NAME
- CLD_APPLIANCE_IDENTIFICATION_ATTR_COMPANY_ID
- CLD_APPLIANCE_IDENTIFICATION_ATTR_BRAND_NAME
- CLD_APPLIANCE_IDENTIFICATION_ATTR_BRAND_ID
- CLD_APPLIANCE_IDENTIFICATION_ATTR_MODEL
- CLD_APPLIANCE_IDENTIFICATION_ATTR_PART_NUMBER

10. Appliance Events and Alerts Cluster

This chapter outlines the Appliance Events and Alerts cluster which is defined in the ZigBee Home Automation profile, and provides an interface for the notification of significant events and alert situations.

The Appliance Events and Alerts cluster has a Cluster ID of 0x0B02.

10.1 Overview

The Appliance Events and Alerts cluster provides an interface for sending notifications of appliance events (e.g. target temperature reached) and alerts (e.g. alarms).

The cluster is enabled by defining CLD_APPLIANCE_EVENTS_AND_ALERTS in the **zcl_options.h** file - see Section 3.5.1. Further compile-time options for the Appliance Events and Alerts cluster are detailed in Section 10.9.

Events are notified in terms of header and event identifier fields (an event may occur when the appliance reaches a certain state, such as the end of its operational cycle).

Alerts are notified in terms of the following fields:

- Alert identification value
- Alert category, one of: Warning, Danger, Failure
- Presence/recovery flag (indicating alert has been either detected or recovered)

10.2 Cluster Structure and Attributes

The Appliance Events and Alerts cluster has no attributes.

10.3 Sending Messages

The Appliance Events and Alerts cluster server resides on the appliance (e.g. a washing machine) and the cluster client resides on a controlling device (normally a remote control unit).

Messages can be sent between the client and the server in the following ways:

- Alerts that are active on the appliance can be requested by the client by sending a ‘Get Alerts’ message to the server (which will reply with a ‘Get Alerts Response’ message)
- Alerts that are active on the appliance can be sent unsolicited from the server to the client in an ‘Alerts Notification’ message
- The server can notify the client of an appliance event by sending an unsolicited ‘Event Notification’ message to the client

Sending the above messages is described in the sub-sections below.

10.3.1 ‘Get Alerts’ Messages from Client to Server

The application on the cluster client can request the alerts that are currently active on the appliance by sending a ‘Get Alerts’ message to the server - this message is sent by calling the function **eCLD_AEAAGetAlertsSend()**. This function returns immediately and the requested alerts are later returned in an E_CLD_APPLIANCE_EVENTS_AND_ALERTS_CMD_GET_ALERTS event, which is generated when a response arrives from the server (Appliance Events and Alerts events are described in Section 10.4).

Note: The cluster server handles the ‘Get Alerts’ message automatically and returns the requested alerts in a ‘Get Alerts Response’ message to the client.

The appliance alerts from the message payload are contained in the above event - for details of this payload and the alert information, refer to Section 10.8.2. Up to 15 alerts can be reported in a single response.

10.3.2 ‘Alerts Notification’ Messages from Server to Client

The cluster server on the appliance can send unsolicited alert notifications to the client in ‘Alerts Notification’ messages. A message of this kind can be sent by the application on the server by calling either of the following functions:

- **eCLD_AEAAAlertsNotificationSend()**
- **eCLD_AEAAGetAlertsResponseORAlertsNotificationSend()**

Note: The latter function is also used internally by the cluster server to send a ‘Get Alerts Response’ message - see Section 10.3.1.

The appliance status information from the ‘Alerts Notification’ message is reported to the application on the cluster client through the event E_CLD_APPLIANCE_EVENTS_AND_ALERTS_CMD_ALERTS_NOTIFICATION, which is generated when the notification arrives from the server (Appliance Events and Alerts events are described in Section 10.4). The appliance alerts from the message payload are contained in the above event - for details of this payload and the alert information, refer to Section 10.8.2. Up to 15 alerts can be reported in a single notification.

10.3.3 'Event Notification' Messages from Server to Client

The cluster server on the appliance can send unsolicited event notifications to the client in 'Event Notification' messages, where each message reports a single appliance event (e.g. oven has reached its target temperature). A message of this kind can be sent by the application on the server by calling the function **eCLD_AEAAEventNotificationSend()**.

The appliance event information from the 'Event Notification' message is reported to the application on the cluster client through the event E_CLD_APPLIANCE_EVENTS_AND_ALERTS_CMD_ALERTS_NOTIFICATION, which is generated when the notification arrives from the server (Appliance Events and Alerts events are described in Section 10.4). The appliance event from the message payload is contained in the above client event - for details of this payload and the embedded appliance event information, refer to Section 10.8.3.

10.4 Appliance Events and Alerts Events

The Appliance Events and Alerts cluster has its own events that are handled through the callback mechanism outlined in Section 4.7 (and fully detailed in the *ZCL User Guide (JN-UG-3077)*). The cluster contains its own event handler. If a device uses this cluster then application-specific Appliance Events and Alerts event handling must be included in the user-defined callback function for the associated endpoint, where this callback function is registered through the relevant endpoint registration function. This callback function will then be invoked when an Appliance Events and Alerts event occurs and needs the attention of the application.

For an Appliance Events and Alerts event, the `eEventType` field of the `tsZCL_CallBackEvent` structure is set to E_ZCL_CBET_CLUSTER_CUSTOM. This event structure also contains an element `sClusterCustomMessage`, which is itself a structure containing a field `pvCustomData`. This field is a pointer to the following `tsCLD_ApplianceEventsAndAlertsCallBackMessage` structure:

```
typedef struct
{
    uint8     u8CommandId
    union
    {
        tsCLD_AEAA_GetAlertsResponseORAlertsNotificationPayload
                                   *psGetAlertsResponseORAlertsNotificationPayload;
        tsCLD_AEAA_EventNotificationPayload
                                   *psEventNotificationPayload;
    } uMessage;
} tsCLD_ApplianceEventsAndAlertsCallBackMessage;
```

When an Appliance Events and Alerts event occurs, one of four command types could have been received. The relevant command type is specified through the `u8CommandId` field of the `tsSM_CallBackMessage` structure. The possible command types are detailed the tables below for events generated on a server and a client.

u8CommandId Enumeration	Description
E_CLD_APPLIANCE_EVENTS_AND_ ALERTS_CMD_GET_ALERTS	A 'Get Alerts' request has been received by the server (appliance)

Table 42: Appliance Events and Alerts Command Types (Events on Server)

u8CommandId Enumeration	Description
E_CLD_APPLIANCE_EVENTS_AND_ ALERTS_CMD_GET_ALERTS	A response to a 'Get Alerts' request has been received by the client, containing the requested alerts (up to 15)
E_CLD_APPLIANCE_EVENTS_AND_ ALERTS_CMD_ALERTS_NOTIFICATION	An 'Alerts Notification' message has been received by the client, containing unsolicited alerts (up to 15)
E_CLD_APPLIANCE_EVENTS_AND_ ALERTS_CMD_EVENT_NOTIFICATION	An 'Event Notification' message has been received by the client

Table 43: Appliance Events and Alerts Command Types (Events on Client)

10.5 Functions

The following Appliance Events and Alerts cluster functions are provided in the HA API:

Function	Page
eCLD_ApplianceEventsAndAlertsCreateApplianceEventsAndAlerts	191
eCLD_AEAAGetAlertsSend	193
eCLD_AEAAGetAlertsResponseORAlertsNotificationSend	194
eCLD_AEAAAlertsNotificationSend	196
eCLD_AEAAEventNotificationSend	197

eCLD_ApplianceEventsAndAlertsCreateApplianceEventsAndAlerts

```
teZCL_Status
eCLD_ApplianceEventsAndAlertsCreateApplianceEventsAndAlerts(
        tsZCL_ClusterInstance *psClusterInstance,
        bool_t bIsServer,
        tsZCL_ClusterDefinition *psClusterDefinition,
        tsCLD_ApplianceEventsAndAlertsCustomDataStructure
                                    *psCustomDataStructure);
```

Description

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

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

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

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

Parameters

psClusterInstance	Pointer to structure containing information about the cluster instance to be created (see the *ZCL User Guide (JN-UG-3077)*). This structure will be updated by the function by initialising individual structure fields.
bIsServer	Type of cluster instance (server or client) to be created: TRUE - server FALSE - client
psClusterDefinition	Pointer to structure indicating the type of cluster to be created (see the *ZCL User Guide (JN-UG-3077)*). In this case, this structure must contain the details of the Appliance Events and Alerts cluster. This parameter can refer to a pre-filled structure called

	`sCLD_ApplianceEventsAndAlerts` which is provided in the **ApplianceEventsAndAlerts.h** file.
pvEndPointSharedStructPtr	Pointer to the shared structure used for attribute storage. This parameter should be the address of the structure of `tsCLD_ApplianceEventsAndAlerts` type which defines the attributes of Appliance Events and Alerts cluster. The function will initialise the attributes with default values.
psCustomDataStructure	Pointer to a structure containing the storage for internal functions of the cluster (see Section 10.8.4)

Returns

E_ZCL_SUCCESS

E_ZCL_FAIL

E_ZCL_ERR_PARAMETER_NULL

E_ZCL_ERR_INVALID_VALUE

eCLD_AEAAGetAlertsSend

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

Description

This function can be used on an Appliance Events and Alerts cluster client to send a 'Get Alerts' message to a cluster server (appliance).

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

Parameter

u8SourceEndPointId	Number of the local endpoint through which to send the message. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values
u8DestinationEndPointId	Number of the endpoint on the remote node to which the message will be sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP
psDestinationAddress	Pointer to a structure holding the address of the node to which the request will be sent
pu8TransactionSequenceNumber	Pointer to a location to receive the Transaction Sequence Number (TSN) of the message

Returns

E_ZCL_SUCCESS

E_ZCL_FAIL

E_ZCL_ERR_PARAMETER_NULL

E_ZCL_ERR_INVALID_VALUE

eCLD_AEAAGetAlertsResponseORAlertsNotificationSend

```
teZCL_Status
eCLD_AEAAGetAlertsResponseORAlertsNotificationSend(
        uint8 u8SourceEndPointId,
        uint8 u8DestinationEndPointId,
        tsZCL_Address *psDestinationAddress,
        uint8 *pu8TransactionSequenceNumber,
        teCLD_ApplianceEventsAndAlerts_CommandId eCommandId,
        tsCLD_AEAA_GetAlertsResponseORAlertsNotificationPayload
                                                    *psPayload);
```

Description

This function can be used on an Appliance Events and Alerts cluster server to send a ‘Get Alerts Response’ message (in reply to a ‘Get Alerts’ message) or an unsolicited ‘Alerts Notification’ message to a cluster client.

The command to be sent must be specified as one of:

- E_CLD_APPLIANCE_EVENTS_AND_ALERTS_CMD_GET_ALERTS
- E_CLD_APPLIANCE_EVENTS_AND_ALERTS_CMD_ALERTS_NOTIFICATION

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

Parameters

u8SourceEndPointId	Number of the local endpoint through which to send the message. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values
u8DestinationEndPointId	Number of the endpoint on the remote node to which the message will be sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP
psDestinationAddress	Pointer to a structure holding the address of the node to which the message will be sent
pu8TransactionSequenceNumber	Pointer to a location to receive the Transaction Sequence Number (TSN) of the message
eCommandId	Enumeration indicating the command to be sent (see above and Section 10.7.1)
psPayload	Pointer to structure containing payload for message (see Section 10.8.2)

Returns

E_ZCL_SUCCESS

E_ZCL_FAIL

E_ZCL_ERR_PARAMETER_NULL

E_ZCL_ERR_INVALID_VALUE

eCLD_AEAAAlertsNotificationSend

```
teZCL_Status eCLD_AEAAAlertsNotificationSend(
        uint8 u8SourceEndPointId,
        uint8 u8DestinationEndPointId,
        tsZCL_Address *psDestinationAddress,
        uint8 *pu8TransactionSequenceNumber,
        tsCLD_AEAA_GetAlertsResponseORAlertsNotificationPayload
                                                        *psPayload);
```

Description

This function can be used on an Appliance Events and Alerts cluster server to send an unsolicited ‘Alerts Notification’ message to a cluster client. The function is an alternative to **eCLD_AEAAGetAlertsResponseORAlertsNotificationSend()**.

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

Parameters

u8SourceEndPointId	Number of the local endpoint through which to send the message. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values
u8DestinationEndPointId	Number of the endpoint on the remote node to which the message will be sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP
psDestinationAddress	Pointer to a structure holding the address of the node to which the message will be sent
pu8TransactionSequenceNumber	Pointer to a location to receive the Transaction Sequence Number (TSN) of the message
psPayload	Pointer to structure containing payload for message (see Section 10.8.2)

Returns

E_ZCL_SUCCESS

E_ZCL_FAIL

E_ZCL_ERR_PARAMETER_NULL

E_ZCL_ERR_INVALID_VALUE

eCLD_AEAAEventNotificationSend

```
teZCL_Status eCLD_AEAAEventNotificationSend(
        uint8 u8SourceEndPointId,
        uint8 u8DestinationEndPointId,
        tsZCL_Address *psDestinationAddress,
        uint8 *pu8TransactionSequenceNumber,
        tsCLD_AEAA_EventNotificationPayload *psPayload);
```

Description

This function can be used on an Appliance Events and Alerts cluster server (appliance) to send an 'Event Notification' message to a cluster client, to indicate that an incident has occurred.

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

Parameters

u8SourceEndPointId	Number of the local endpoint through which to send the message. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values
u8DestinationEndPointId	Number of the endpoint on the remote node to which the message will be sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP
psDestinationAddress	Pointer to a structure holding the address of the node to which the message will be sent
pu8TransactionSequenceNumber	Pointer to a location to receive the Transaction Sequence Number (TSN) of the message
psPayload	Pointer to structure containing payload for message (see Section 10.8.3)

Returns

E_ZCL_SUCCESS

E_ZCL_FAIL

E_ZCL_ERR_PARAMETER_NULL

E_ZCL_ERR_INVALID_VALUE

10.6 Return Codes

The Appliance Events and Alerts cluster functions use the ZCL return codes defined in the *ZCL User Guide (JN-UG-3077)*.

10.7 Enumerations

10.7.1 'Command ID' Enumerations

The following enumerations are used in commands received on a cluster server or client.

```
typedef enum PACK
{
   E_CLD_APPLIANCE_EVENTS_AND_ALERTS_CMD_GET_ALERTS = 0x00,
   E_CLD_APPLIANCE_EVENTS_AND_ALERTS_CMD_ALERTS_NOTIFICATION,
   E_CLD_APPLIANCE_EVENTS_AND_ALERTS_CMD_EVENT_NOTIFICATION
} teCLD_ApplianceEventsAndAlerts_CommandId;
```

The above enumerations are described in the table below.

Enumeration	Description
E_CLD_APPLIANCE_EVENTS_AND_ALERTS_CMD_GET_ALERTS	'Get Alerts' request (on server) or response (on client)
E_CLD_APPLIANCE_EVENTS_AND_ALERTS_CMD_ALERTS_NOTIFICATION	Alerts notification (on client)
E_CLD_APPLIANCE_EVENTS_AND_ALERTS_CMD_EVENT_NOTIFICATION	Events notification (on server)

Table 44: 'Command ID' Enumerations

10.8 Structures

10.8.1 tsCLD_ApplianceEventsAndAlertsCallBackMessage

For an Appliance Events and Alerts event, the `eEventType` field of the `tsZCL_CallBackEvent` structure is set to E_ZCL_CBET_CLUSTER_CUSTOM. This event structure also contains an element `sClusterCustomMessage`, which is itself a structure containing a field `pvCustomData`. This field is a pointer to the following `tsCLD_ApplianceEventsAndAlertsCallBackMessage` structure:

```
typedef struct
{
    uint8     u8CommandId
    union
    {
        tsCLD_AEAA_GetAlertsResponseORAlertsNotificationPayload
                                   *psGetAlertsResponseORAlertsNotificationPayload;
        tsCLD_AEAA_EventNotificationPayload
                                   *psEventNotificationPayload;
    } uMessage;
} tsCLD_ApplianceEventsAndAlertsCallBackMessage;
```

where:

- `u8CommandId` indicates the type of Appliance Events and Alerts command that has been received, one of:
  - E_CLD_APPLIANCE_EVENTS_AND_ALERTS_CMD_GET_ALERTS
  - E_CLD_APPLIANCE_EVENTS_AND_ALERTS_CMD_ALERTS_NOTIFICATION
  - E_CLD_APPLIANCE_EVENTS_AND_ALERTS_CMD_EVENT_NOTIFICATION
- `uMessage` is a union containing the command payload as one of (depending on the value of `u8CommandId`):
  - `psGetAlertsResponseORAlertsNotificationPayload` is a pointer to the payload of an “Get Alerts” response message or an alerts notification message (see Section 10.8.2)
  - `psEventNotificationPayload` is a pointer to the payload of an events notification message (see Section 10.8.3)

10.8.2 tsCLD_AEAA_GetAlertsResponseORAlertsNotificationPayload

This structure contains the payload for a ‘Get Alerts Response’ message or an ‘Alerts Notification’ message.

```
typedef struct
{
   zuint8  u8AlertsCount;
   zuint24 au24AlertStructure[
             CLD_APPLIANCE_EVENTS_AND_ALERTS_MAXIMUM_NUM_OF_ALERTS];
} tsCLD_AEAA_GetAlertsResponseORAlertsNotificationPayload;
```

where:

- `u8AlertsCount` is an 8-bit bitmap containing the following alerts information:

Bits	Description
0-3	Number of reported alerts
4-7	Type of alert: • 0x0: Unstructured • 0x1-0xF: Reserved

- `au24AlertStructure[]` is an array of 24-bit bitmaps, with one bitmap for each reported alert, containing the following information:

Bits	Description
0-7	Alert ID: • 0x0: Reserved • 0x01-0x3F: Standardised • 0x40-0x7F: Non-standardised • 0x80-0xFF: Proprietary
8-11	Category: • 0x0: Reserved • 0x1: Warning • 0x2: Danger • 0x3: Failure • 0x4–0xF: Reserved
12-13	Presence or recovery: • 0x0: Presence (alert detected) • 0x1: Recovery (alert recovered) • 0x2–0x3: Reserved
14-15	Reserved (set to 0x0)
16-23	Non-standardised or proprietary

10.8.3 tsCLD_AEAA_EventNotificationPayload

This structure contains the payload for an ‘Event Notification’ message.

```
typedef struct
{
        zuint8                          u8EventHeader;
        zuint8                          u8EventIdentification;
} tsCLD_AEAA_EventNotificationPayload;
```

where:

- `u8EventHeader` is reserved and set to 0
- `u8EventIdentification` is the identifier of the event being notified:
  - 0x01: End of operational cycle reached
  - 0x02: Reserved
  - 0x03: Reserved
  - 0x04: Target temperature reached
  - 0x05: End of cooking process reached
  - 0x06: Switching off
  - 0xF7: Wrong data

  (Values 0x00 to 0x3F are standardised, 0x40 to 0x7F are non-standardised, and 0x80 to 0xFF except 0xF7 are proprietary)

10.8.4 tsCLD_ApplianceEventsAndAlertsCustomDataStructure

The Appliance Events and Alerts cluster requires extra storage space to be allocated to be used by internal functions. The structure definition for this storage is shown below:

```
typedef struct
{
    tsZCL_ReceiveEventAddress sReceiveEventAddress;
    tsZCL_CallBackEvent sCustomCallBackEvent;
    tsCLD_ApplianceEventsAndAlertsCallBackMessage sCallBackMessage;
} tsCLD_ApplianceEventsAndAlertsCustomDataStructure;
```

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

10.9 Compile-Time Options

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

To enable the Appliance Events and Alerts cluster in the code to be built, it is necessary to add the following line to the file:

```
#define CLD_APPLIANCE_EVENTS_AND_ALERTS
```

In addition, to enable the cluster as a client or server, it is also necessary to add one of the following lines to the same file:

```
#define APPLIANCE_EVENTS_AND_ALERTS_SERVER
#define APPLIANCE_EVENTS_AND_ALERTS_CLIENT
```

Maximum Number of Alerts Reported

The maximum number of alerts that can be reported in a response or notification can be defined (as n) using the following definition in the **zcl_options.h** file:

```
#define CLD_APPLIANCE_EVENTS_AND_ALERTS_MAXIMUM_NUM_OF_ALERTS n
```

The default value is 16, which is the upper limit on this value, and n must therefore not be greater than 16.

11. Appliance Statistics Cluster

This chapter outlines the Appliance Statistics cluster which is defined in the ZigBee Home Automation profile, and provides an interface for supplying statistical information about an appliance.

The Appliance Statistics cluster has a Cluster ID of 0x0B03.

11.1 Overview

The Appliance Statistics cluster provides an interface for sending appliance statistics in the form of data logs to a collector node, which may be a gateway.

The cluster is enabled by defining CLD_APPLIANCE_STATISTICS in the **zcl_options.h** file - see Section 3.5.1. Further compile-time options for the Appliance Statistics cluster are detailed in Section 11.10.

The cluster client may obtain logs from the server (appliance) in any of the following ways:

- Unsolicited log notifications sent by the server
- Solicited responses obtained by:
  - Client sending 'Log Queue Request' to enquire whether logs are available
  - Client sending 'Log Request' for each log available
- Semi-solicited responses obtained by:
  - Server sending 'Statistics Available' notification to indicate that logs are available
  - Client sending 'Log Request' for each log available

11.2 Cluster Structure and Attributes

The Appliance Statistics cluster is contained in the following `tsCLD_ApplianceStatistics` structure:

```
typedef struct
{
    zuint32       u32LogMaxSize;
    zuint8        u8LogQueueMaxSize;
}tsCLD_ApplianceStatistics;
```

where:

- `u32LogMaxSize` is a mandatory attribute which specifies the maximum size, in bytes, of the payload of a log notification and log response. This value should not be greater than 70 bytes (otherwise the Partition cluster is needed)
- `u8LogQueueMaxSize` is a mandatory attribute which specifies the maximum number of logs in the queue on the cluster server that are available to be requested by the client

11.3 Sending Messages

The Appliance Statistics cluster server resides on the appliance (e.g. a washing machine) and the cluster client resides on a controlling device (normally a remote control unit).

Messages can be sent between the client and the server in the following ways:

- The client can enquire whether any data logs are available on the appliance (server) by sending a 'Log Queue Request' to the server (which will reply with a 'Log Queue Response' message)
- The server can notify the client that data logs are available by sending an unsolicited 'Statistics Available' message to the client
- The client can request a current data log from the appliance (server) by sending a 'Log Request' message to the server (which will reply with a 'Log Response' message)
- The server can send an unsolicited data log to the client in a 'Log Notification' message

Sending the above messages is described in the sub-sections below.

11.3.1 'Log Queue Request' Messages from Client to Server

The application on the cluster client can enquire about the availability of data logs on the appliance by sending a 'Log Queue Request' message to the server. This message is sent by calling the function **eCLD_ASCLogQueueRequestSend()**. This function returns immediately and the log availability is later returned in an E_CLD_APPLIANCE_STATISTICS_CMD_LOG_QUEUE_RESPONSE event, which is generated when a response arrives from the server (Appliance Statistics events are described in Section 11.5).

Note: The cluster server handles the 'Log Queue Request' message automatically and returns the requested information in a 'Log Queue Response' message to the client.

The log availability information from the message payload is contained in the above event, and comprises the number of logs currently in the log queue and their log IDs - for details of this payload and the availability information, refer to Section 11.9.4.

11.3.2 'Statistics Available' Messages from Server to Client

The cluster server can notify the client when data logs are available by sending an unsolicited 'Statistics Available' message to the client. This message contains the number of logs in the log queue and the log IDs. A message of this kind can be sent by the application on the server by calling either of the following functions:

- **eCLD_ASCStatisticsAvailableSend()**
- **eCLD_ASCLogQueueResponseORStatisticsAvailableSend()**

Note 1: The latter function is also used internally by the cluster server to send a 'Log Queue Response' message - see Section 11.3.1.

Note 2: Before calling either function, the relevant log(s) should be added to the local log queue as described in Section 11.4.1. This is because the logs need to be in the queue to allow the server to perform further actions on them - for example, to process a 'Log Request'.

The log availability information from the 'Statistics Available' message is reported to the application on the cluster client through the event E_CLD_APPLIANCE_STATISTICS_CMD_STATISTICS_AVAILABLE, which is generated when the message arrives from the server (Appliance Statistics events are described in Section 11.5). The availability information from the message payload is contained in the above event - for details of this payload and the availability information, refer to Section 11.9.4.

11.3.3 ‘Log Request’ Messages from Client to Server

The application on the cluster client can request the log with a particular log ID from the appliance by sending a ‘Log Request’ message to the server. This message is sent by calling the function **eCLD_ASCLogRequestSend()**. This function returns immediately and the requested log information is later returned in an E_CLD_APPLIANCE_STATISTICS_CMD_LOG_QUEUE_RESPONSE event, which is generated when a response arrives from the server (Appliance Statistics events are described in Section 11.5).

Note 1: This function should normally be called after a ‘Log Queue Response’ or ‘Statistics Available’ message has been received by the client, indicating that logs are available on the server.

Note 2: The cluster server handles the ‘Log Request’ message automatically and returns the requested log information in a ‘Log Response’ message to the client.

The log information from the message payload is contained in the above event - for details of this payload and the supplied log information, refer to Section 11.9.3.

11.3.4 ‘Log Notification’ Messages from Server to Client

The cluster server can supply the client with an individual data log by sending an unsolicited ‘Log Notification’ message to the client. This message is sent by the application on the server by calling either of the following functions:

- **eCLD_ASCLogNotificationSend()**
- **eCLD_ASCLogNotificationORLogResponseSend()**

Note 1: The latter function is also used internally by the cluster server to send a ‘Log Response’ message - see Section 11.3.1.

Note 2: Before calling either function, the relevant log should be added to the local log queue as described in Section 11.4.1. This is because the log needs to be in the queue to allow the server to perform further actions on it - for example, to process a ‘Log Request’. Refer to the Example below.

The log information from the ‘Log Notification’ message is reported to the application on the cluster client through the event E_CLD_APPLIANCE_STATISTICS_CMD_LOG_NOTIFICATION, which is generated when the message arrives from the server (Appliance Statistics events are described in Section 11.5). The log information from the message payload is contained in the above event - for details of this payload and the supplied log information, refer to Section 11.9.3.

Example

To allow a new log to be available for later processing, it should be added to the local log queue on the server before a ‘Log Notification’ message is sent, as follows:

1. Add the log to the queue by calling the function **eCLD_ASCAddLog()**.
2. At a later point when the notification is to be sent, obtain the log from the queue using the function **eCLD_ASCGetLogEntry()** - see Section 11.4.2.
3. Send the notification, e.g. by calling **eCLD_ASCLogNotificationSend()**.

11.4 Log Operations on Server

Appliance Statistics cluster functions are provided in the HA API to allow the application on the cluster server (appliance) to perform the following local log operations:

- Add a log to the log queue
- Remove a log from the log queue
- Obtain a list of the logs in the log queue
- Obtain an individual log from the log queue

These operations are described in the sub-sections below.

11.4.1 Adding and Removing Logs

A data log can be added to the local log queue (on the cluster server) using the function **eCLD_ASCAddLog()**. The log must be given an identifier and the UTC time at which the log was added must be specified. The length of the log, in bytes, must be less than the value of CLD_APPLIANCE_STATISTICS_ATTR_LOG_MAX_SIZE, which is defined in the **zcl_options.h** files (and must be less than or equal to 70).

An existing log can be removed from the local log queue using the function **eCLD_ASCRemoveLog()**. The log is specified using its identifier.

11.4.2 Obtaining Logs

A list of the logs that are currently in the local log queue (on the cluster server) can be obtained by calling the function **eCLD_ASCGetLogsAvailable()**. This function provides the number of logs in the queue and a list of the log identifiers.

An individual log from the local log queue can be obtained using the function **eCLD_ASCGetLogEntry()**. The required log is specified by means of its identifier.

Normally, **eCLD_ASCGetLogsAvailable()** is called first to obtain a list of the available logs and then **eCLD_ASCGetLogEntry()** is called for each log.

11.5 Appliance Statistics Events

The Appliance Statistics cluster has its own events that are handled through the callback mechanism outlined in Section 4.7 (and fully detailed in the *ZCL User Guide (JN-UG-3077)*). The cluster contains its own event handler. If a device uses this cluster then application-specific Appliance Statistics event handling must be included in the user-defined callback function for the associated endpoint, where this callback function is registered through the relevant endpoint registration function. This callback function will then be invoked when an Appliance Statistics event occurs and needs the attention of the application.

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

```
typedef struct
{
    uint8    u8CommandId;
    union
    {
        tsCLD_ASC_LogNotificationORLogResponsePayload
                                      *psLogNotificationORLogResponsePayload;
        tsCLD_ASC_LogQueueResponseORStatisticsAvailablePayload
                                      *psLogQueueResponseORStatisticsAvailabePayload;
        tsCLD_ASC_LogRequestPayload   *psLogRequestPayload;
    } uMessage;
} tsCLD_ApplianceStatisticsCallBackMessage;
```

When an Appliance Statistics event occurs, one of four command types could have been received. The relevant command type is specified through the `u8CommandId` field of the `tsSM_CallBackMessage` structure. The possible command types are detailed the tables below for events generated on a server and a client.

u8CommandId Enumeration	Description
E_CLD_APPLIANCE_STATISTICS_CMD_LOG_REQUEST	A ‘Log Request’ message has been received by the server (appliance)
E_CLD_APPLIANCE_STATISTICS_CMD_LOG_QUEUE_REQUEST	A ‘Log Queue Request’ message has been received by the server (appliance)

Table 45: Appliance Statistics Command Types (Events on Server)

u8CommandId Enumeration	Description
E_CLD_APPLIANCE_STATISTICS_CMD_LOG_NOTIFICATION	A ‘Log Notification’ message has been received by the client
E_CLD_APPLIANCE_STATISTICS_CMD_LOG_RESPONSE	A ‘Log Response’ message has been received by the client

Table 46: Appliance Statistics Command Types (Events on Client)

u8CommandId Enumeration	Description
E_CLD_APPLIANCE_STATISTICS_ CMD_LOG_QUEUE_RESPONSE	A ‘Log Queue Response’ message has been received by the client
E_CLD_APPLIANCE_STATISTICS_ CMD_STATISTICS_AVAILABLE	A ‘Statistics Available’ message has been received by the client

Table 46: Appliance Statistics Command Types (Events on Client)

11.6 Functions

The following Appliance Statistics cluster functions are provided in the HA API:

Function	**Page**
eCLD_ApplianceStatisticsCreateApplianceStatistics	210
eCLD_ASCAddLog	212
eCLD_ASCRemoveLog	213
eCLD_ASCGetLogsAvailable	214
eCLD_ASCGetLogEntry	215
eCLD_ASCLogQueueRequestSend	216
eCLD_ASCLogRequestSend	217
eCLD_ASCLogQueueResponseORStatisticsAvailableSend	218
eCLD_ASCStatisticsAvailableSend	220
eCLD_ASCLogNotificationORLogResponseSend	221
eCLD_ASCLogNotificationSend	223

eCLD_ApplianceStatisticsCreateApplianceStatistics

teZCL_Status
eCLD_ApplianceStatisticsCreateApplianceStatistics(
tsZCL_ClusterInstance ***psClusterInstance*,**
bool_t *bIsServer*,
tsZCL_ClusterDefinition ***psClusterDefinition*,**
void ***pvEndPointSharedStructPtr*,**
uint8 ***pu8AttributeControlBits*,**
tsCLD_ApplianceStatisticsCustomDataStructure
***psCustomDataStructure*);**

Description

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

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

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

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

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

The array declaration should be as follows:

```
uint8 au8AppApplianceStatisticsClusterAttributeControlBits[
CLD_APPLIANCE_STATISTICS_MAX_NUMBER_OF_ATTRIBUTE];
```

The function will initialise the array elements to zero.

Parameters

psClusterInstance	Pointer to structure containing information about the cluster instance to be created (see the *ZCL User Guide (JN-UG-3077)*). This structure will be updated by the function by initialising individual structure fields.
bIsServer	Type of cluster instance (server or client) to be created: TRUE - server FALSE - client
psClusterDefinition	Pointer to structure indicating the type of cluster to be created (see the *ZCL User Guide (JN-UG-3077)*). In this case, this structure must contain the details of the Appliance Statistics cluster. This parameter can refer to a pre-filled structure called `sCLD_ApplianceStatistics` which is provided in the **ApplianceStatistics.h** file.
pvEndPointSharedStructPtr	Pointer to the shared structure used for attribute storage. This parameter should be the address of the structure of type `tsCLD_ApplianceStatistics` which defines the attributes of Appliance Statistics cluster. The function will initialise the attributes with default values.
pu8AttributeControlBits	Pointer to an array of **uint8** values, with one element for each attribute in the cluster (see above).
psCustomDataStructure	Pointer to a structure containing the storage for internal functions of the cluster (see Section 11.9.6)

Returns

E_ZCL_SUCCESS

E_ZCL_FAIL

E_ZCL_ERR_PARAMETER_NULL

E_ZCL_ERR_INVALID_VALUE

eCLD_ASCAddLog

```
teZCL_CommandStatus eCLD_ASCAddLog(
                                uint8 u8SourceEndPointId,
                                uint32 u32LogId,
                                uint8 u8LogLength,
                                uint32 u32Time,
                                uint8 *pu8LogData);
```

Description

This function can be used on an Appliance Statistics cluster server to add a data log to the log queue.

The length of the data log, in bytes, must be less than the defined value of CLD_APPLIANCE_STATISTICS_ATTR_LOG_MAX_SIZE (which must be less than or equal to 70).

Parameter

u8SourceEndPointId	Number of the local endpoint on which the Appliance Statistics cluster server resides
u32LogId	Identifier of log
u8LogLength	Length of log, in bytes
u32Time	UTC time at which log was produced
pu8LogData	Pointer to log data

Returns

E_ZCL_CMDS_SUCCESS

E_ZCL_CMDS_FAIL

E_ZCL_CMDS_INVALID_VALUE (log too long)

E_ZCL_CMDS_INVALID_FIELD (NULL pointer to log data)

E_ZCL_CMDS_INSUFFICIENT_SPACE

eCLD_ASCRemoveLog

```
teZCL_CommandStatus eCLD_ASCRemoveLog(
                        uint8 u8SourceEndPointId,
                        uint32 u32LogId);
```

Description

This function can be used on an Appliance Statistics cluster server to remove the specified data log from the log queue.

Parameter

u8SourceEndPointId	Number of the local endpoint on which the Appliance Statistics cluster server resides
u32LogId	Identifier of log

Returns

E_ZCL_CMDS_SUCCESS

E_ZCL_CMDS_FAIL

eCLD_ASCGetLogsAvailable

```
teZCL_CommandStatus eCLD_ASCGetLogsAvailable(
                              uint8 u8SourceEndPointId,
                              uint32 *pu32LogId,
                              uint8 *pu8LogIdCount);
```

Description

This function can be used on an Appliance Statistics cluster server to obtain a list of the data logs in the log queue. The number of available logs and a list of their log IDs will be obtained.

Parameter

u8SourceEndPointId	Number of the local endpoint on which the Appliance Statistics cluster server resides
pu32LogId	Pointer to an area of memory to receive the list of 32-bit log IDs
pu8LogIdCount	Pointer to an area of memory to receive the number of logs in the queue

Returns

E_ZCL_CMDS_SUCCESS
E_ZCL_CMDS_FAIL

eCLD_ASCGetLogEntry

```
teZCL_CommandStatus eCLD_ASCGetLogEntry(
                          uint8 u8SourceEndPointId,
                          uint32 u32LogId,
                          tsCLD_LogTable **ppsLogTable);
```

Description

This function can be used on an Appliance Statistics cluster server to obtain the data log with the specified log ID.

Parameter

u8SourceEndPointId	Number of the local endpoint on which the Appliance Statistics cluster server resides
u32LogId	Log ID of the required data log
ppsLogTable	Pointer to a memory location to receive a pointer to the required data log

Returns

E_ZCL_CMDS_SUCCESS

E_ZCL_CMDS_FAIL

E_ZCL_CMDS_NOT_FOUND (specified log not present)

eCLD_ASCLogQueueRequestSend

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

Description

This function can be used on an Appliance Statistics cluster client to send a ‘Log Queue Request’ message to a cluster server (appliance), in order enquire about the availability of logs on the server.

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

Parameters

u8SourceEndPointId	Number of the local endpoint through which to send the message. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values
u8DestinationEndPointId	Number of the endpoint on the remote node to which the message will be sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP
psDestinationAddress	Pointer to a structure holding the address of the node to which the message will be sent
pu8TransactionSequenceNumber	Pointer to a location to receive the Transaction Sequence Number (TSN) of the message

Returns

E_ZCL_SUCCESS

E_ZCL_FAIL

E_ZCL_ERR_PARAMETER_NULL

E_ZCL_ERR_INVALID_VALUE

eCLD_ASCLogRequestSend

```
teZCL_Status eCLD_ASCLogRequestSend(
                    uint8 u8SourceEndPointId,
                    uint8 u8DestinationEndPointId,
                    tsZCL_Address *psDestinationAddress,
                    uint8 *pu8TransactionSequenceNumber,
                    tsCLD_ASC_LogRequestPayload *psPayload);
```

Description

This function can be used on an Appliance Statistics cluster client to send a 'Log Request' message to a cluster server (appliance), in order request the data log with a specified log ID.

The function should normally be called after enquiring about log availability using the function **eCLD_ASCLogQueueRequestSend()** or after receiving an unsolicited 'Statistics Available' notification from the server.

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

Parameters

u8SourceEndPointId	Number of the local endpoint through which to send the message. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values
u8DestinationEndPointId	Number of the endpoint on the remote node to which the message will be sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP
psDestinationAddress	Pointer to a structure holding the address of the node to which the message will be sent
pu8TransactionSequenceNumber	Pointer to a location to receive the Transaction Sequence Number (TSN) of the message
psPayload	Pointer to *a* structure containing the payload for the 'Log Request', including the relevant log ID (see Section 11.9.2)

Returns

E_ZCL_SUCCESS

E_ZCL_FAIL

E_ZCL_ERR_PARAMETER_NULL

E_ZCL_ERR_INVALID_VALUE

eCLD_ASCLogQueueResponseORStatisticsAvailableSend

```
teZCL_Status
eCLD_ASCLogQueueResponseORStatisticsAvailableSend(
            uint8 u8SourceEndPointId,
            uint8 u8DestinationEndPointId,
            tsZCL_Address *psDestinationAddress,
            uint8 *pu8TransactionSequenceNumber,
            teCLD_ApplianceStatistics_ServerCommandId
                                                  eCommandId);
```

Description

This function can be used on an Appliance Statistics cluster server to send a 'Log Queue Response' message (in reply to a 'Log Queue Request' message) or an unsolicited 'Statistics Available' message to a cluster client.

The command to be sent must be specified as one of:

- E_CLD_APPLIANCE_STATISTICS_CMD_LOG_QUEUE_RESPONSE
- E_CLD_APPLIANCE_STATISTICS_CMD_STATISTICS_AVAILABLE

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

Parameters

u8SourceEndPointId	Number of the local endpoint through which to send the message. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values
u8DestinationEndPointId	Number of the endpoint on the remote node to which the message will be sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP
psDestinationAddress	Pointer to a structure holding the address of the node to which the message will be sent
pu8TransactionSequenceNumber	Pointer to a location to receive the Transaction Sequence Number (TSN) of the message
eCommandId	Enumeration indicating the command to be sent (see above and Section 11.8.3)

Returns

E_ZCL_SUCCESS

E_ZCL_FAIL

E_ZCL_ERR_PARAMETER_NULL

E_ZCL_ERR_INVALID_VALUE

eCLD_ASCStatisticsAvailableSend

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

Description

This function can be used on an Appliance Statistics cluster server to send an unsolicited 'Statistics Available' message to a cluster client. The function is an alternative to **eCLD_ASCLogQueueResponseORStatisticsAvailableSend()**.

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

Parameters

u8SourceEndPointId	Number of the local endpoint through which to send the message. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values
u8DestinationEndPointId	Number of the endpoint on the remote node to which the message will be sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP
psDestinationAddress	Pointer to a structure holding the address of the node to which the message will be sent
pu8TransactionSequenceNumber	Pointer to a location to receive the Transaction Sequence Number (TSN) of the message

Returns

E_ZCL_SUCCESS

E_ZCL_FAIL

E_ZCL_ERR_PARAMETER_NULL

E_ZCL_ERR_INVALID_VALUE

eCLD_ASCLogNotificationORLogResponseSend

```
teZCL_Status
eCLD_ASCLogNotificationORLogResponseSend(
    uint8 u8SourceEndPointId,
    uint8 u8DestinationEndPointId,
    tsZCL_Address *psDestinationAddress,
    uint8 *pu8TransactionSequenceNumber,
    teCLD_ApplianceStatistics_ServerCommandId
                                            eCommandId,
    tsCLD_ASC_LogNotificationORLogResponsePayload
                                            *psPayload);
```

Description

This function can be used on an Appliance Statistics cluster server to send a 'Log Response' message (in reply to a 'Log Request' message) or an unsolicited 'Log Notification' message to a cluster client.

The command to be sent must be specified as one of:

- E_CLD_APPLIANCE_STATISTICS_CMD_LOG_NOTIFICATION
- E_CLD_APPLIANCE_STATISTICS_CMD_LOG_RESPONSE

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

Parameters

u8SourceEndPointId	Number of the local endpoint through which to send the message. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values
u8DestinationEndPointId	Number of the endpoint on the remote node to which the message will be sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP
psDestinationAddress	Pointer to a structure holding the address of the node to which the message will be sent
pu8TransactionSequenceNumber	Pointer to a location to receive the Transaction Sequence Number (TSN) of the message
eCommandId	Enumeration indicating the command to be sent (see above and Section 11.8.3)
psPayload	Pointer to structure containing payload for message (see Section 11.9.3)

Returns

E_ZCL_SUCCESS
E_ZCL_FAIL
E_ZCL_ERR_PARAMETER_NULL
E_ZCL_ERR_INVALID_VALUE

eCLD_ASCLogNotificationSend

```
teZCL_Status eCLD_ASCLogNotificationSend(
        uint8 u8SourceEndPointId,
        uint8 u8DestinationEndPointId,
        tsZCL_Address *psDestinationAddress,
        uint8 *pu8TransactionSequenceNumber,
        tsCLD_ASC_LogNotificationORLogResponsePayload
                                                 *psPayload);
```

Description

This function can be used on an Appliance Statistics cluster server to send an unsolicited 'Log Notification' message to a cluster client. The function is an alternative to **eCLD_ASCLogNotificationORLogResponseSend()**.

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

Parameters

u8SourceEndPointId	Number of the local endpoint through which to send the message. This parameter is used both to send the message and to identify the instance of the shared structure holding the required attribute values
u8DestinationEndPointId	Number of the endpoint on the remote node to which the message will be sent. This parameter is ignored when sending to address types eZCL_AMBOUND and eZCL_AMGROUP
psDestinationAddress	Pointer to a structure holding the address of the node to which the message will be sent
pu8TransactionSequenceNumber	Pointer to a location to receive the Transaction Sequence Number (TSN) of the message
psPayload	Pointer to structure containing payload for message (see Section 11.9.3)

Returns

E_ZCL_SUCCESS

E_ZCL_FAIL

E_ZCL_ERR_PARAMETER_NULL

E_ZCL_ERR_INVALID_VALUE

11.7 Return Codes

The Appliance Statistics cluster functions use the ZCL return codes defined in the *ZCL User Guide (JN-UG-3077)*.

11.8 Enumerations

11.8.1 ‘Attribute ID’ Enumerations

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

```
typedef enum PACK
{
    E_CLD_APPLIANCE_STATISTICS_ATTR_ID_LOG_MAX_SIZE = 0x0000,
    E_CLD_APPLIANCE_STATISTICS_ATTR_ID_LOG_QUEUE_MAX_SIZE
} teCLD_ApplianceStatistics_Cluster_AttrID;
```

11.8.2 ‘Client Command ID’ Enumerations

The following enumerations are used in commands issued on a cluster client.

```
typedef enum PACK
{
    E_CLD_APPLIANCE_STATISTICS_CMD_LOG_REQUEST = 0x00,
    E_CLD_APPLIANCE_STATISTICS_CMD_LOG_QUEUE_REQUEST
} teCLD_ApplianceStatistics_ClientCommandId;
```

The above enumerations are described in the table below.

Enumeration	**Description**
E_CLD_APPLIANCE_STATISTICS_CMD_LOG_REQUEST	‘Log Request’ message
E_CLD_APPLIANCE_STATISTICS_CMD_LOG_QUEUE_REQUEST	‘Log Queue Request’ message

Table 47: ‘Client Command ID’ Enumerations

11.8.3 'Server Command ID' Enumerations

The following enumerations are used in commands issued on a cluster server.

```
typedef enum PACK
{
    E_CLD_APPLIANCE_STATISTICS_CMD_LOG_NOTIFICATION = 0x00,
    E_CLD_APPLIANCE_STATISTICS_CMD_LOG_RESPONSE,
    E_CLD_APPLIANCE_STATISTICS_CMD_LOG_QUEUE_RESPONSE,
    E_CLD_APPLIANCE_STATISTICS_CMD_STATISTICS_AVAILABLE
} teCLD_ApplianceStatistics_ServerCommandId;
```

The above enumerations are described in the table below.

Enumeration	Description
E_CLD_APPLIANCE_STATISTICS_CMD_LOG_NOTIFICATION	A 'Log Notification' message
E_CLD_APPLIANCE_STATISTICS_CMD_LOG_RESPONSE	A 'Log Response' message
E_CLD_APPLIANCE_STATISTICS_CMD_LOG_QUEUE_RESPONSE	A 'Log Queue Response' message
E_CLD_APPLIANCE_STATISTICS_CMD_STATISTICS_AVAILABLE	A 'Statistics Available' message

Table 48: 'Server Command ID' Enumerations

11.9 Structures

11.9.1 tsCLD_ApplianceStatisticsCallBackMessage

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

```
typedef struct
{
    uint8    u8CommandId;
    union
    {
       tsCLD_ASC_LogNotificationORLogResponsePayload
                                    *psLogNotificationORLogResponsePayload;
       tsCLD_ASC_LogQueueResponseORStatisticsAvailablePayload
                                    *psLogQueueResponseORStatisticsAvailabePayload;
       tsCLD_ASC_LogRequestPayload *psLogRequestPayload;
    } uMessage;
} tsCLD_ApplianceStatisticsCallBackMessage;
```

where:

- u8CommandId indicates the type of Appliance Statistics command that has been received, one of:
    - E_CLD_APPLIANCE_STATISTICS_CMD_LOG_REQUEST
    - E_CLD_APPLIANCE_STATISTICS_CMD_LOG_QUEUE_REQUEST
    - E_CLD_APPLIANCE_STATISTICS_CMD_LOG_NOTIFICATION
    - E_CLD_APPLIANCE_STATISTICS_CMD_LOG_RESPONSE
    - E_CLD_APPLIANCE_STATISTICS_CMD_LOG_QUEUE_RESPONSE
    - E_CLD_APPLIANCE_STATISTICS_CMD_STATISTICS_AVAILABLE
- uMessage is a union containing the command payload as one of (depending on the value of u8CommandId):
    - psLogNotificationORLogResponsePayload is a pointer to the payload of a 'Log Notification' or 'Log Response' message (see Section 11.9.3)
    - psLogQueueResponseORStatisticsAvailabePayload is a pointer to the payload of a 'Log Queue Response' or 'Statistics Available' message (see Section 11.9.4)
    - psLogRequestPayload is a pointer to the payload of a 'Log Request' message (see Section 11.9.2)

11.9.2 tsCLD_ASC_LogRequestPayload

This structure contains the payload for the 'Log Request' message.

```
typedef struct
{
    zuint32   u32LogId;
} tsCLD_ASC_LogRequestPayload;
```

where u32LogId is the identifier of the data log being requested.

11.9.3 tsCLD_ASC_LogNotificationORLogResponsePayload

This structure contains the payload for the 'Log Notification' and 'Log Response' messages.

```
typedef struct
{
    zutctime    utctTime;
    zuint32     u32LogId;
    zuint32     u32LogLength;
    uint8      *pu8LogData;
} tsCLD_ASC_LogNotificationORLogResponsePayload;
```

where:

- utctTime is the UTC time at which the reported log was produced
- u32LogId is the identifier of the reported log
- u32LogLength is the length, in bytes, of the reported log
- pu8LogData is a pointer to an area of memory to receive the data of the reported log

11.9.4 tsCLD_ASC_LogQueueResponseORStatisticsAvailablePayload

This structure contains the payload for the 'Log Queue Response' and 'Statistics Available' messages.

```
typedef struct
{
    zuint8     u8LogQueueSize;
    zuint32   *pu32LogId;
} tsCLD_ASC_LogQueueResponseORStatisticsAvailablePayload;
```

where:

- u8LogQueueSize indicates the number of logs currently in the log queue
- pu32LogId is a pointer to an area of memory to receive the sequence of 32-bit log IDs of the logs in the queue

11.9.5 tsCLD_LogTable

This structure is used to store the details of a data log.

```
typedef struct
{
    zutctime    utctTime;
    uint32      u32LogID;
    uint8       u8LogLength;
    uint8      *pu8LogData;
} tsCLD_LogTable;
```

where:

- utctTime is the UTC time at which the log was produced
- u32LogId is the identifier of the log
- u32LogLength is the length, in bytes, of the log
- pu8LogData is a pointer to an area of memory to receive the data of the log

11.9.6 tsCLD_ApplianceStatisticsCustomDataStructure

The Appliance Statistics cluster requires extra storage space to be allocated to be used by internal functions. The structure definition for this storage is shown below:

```
typedef struct
{
    tsZCL_ReceiveEventAddress sReceiveEventAddress;
    tsZCL_CallBackEvent sCustomCallBackEvent;
    tsCLD_ApplianceStatisticsCallBackMessage sCallBackMessage;
#if (defined CLD_APPLIANCE_STATISTICS) && (defined APPLIANCE_STATISTICS_SERVER)
    tsCLD_LogTable asLogTable[CLD_APPLIANCE_STATISTICS_ATTR_LOG_QUEUE_MAX_SIZE];
#endif
} tsCLD_ApplianceStatisticsCustomDataStructure;
```

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

11.10 Compile-Time Options

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

To enable the Appliance Statistics cluster in the code to be built, it is necessary to add the following line to the file:

```
#define CLD_APPLIANCE_STATISTICS
```

In addition, to enable the cluster as a client or server, it is also necessary to add one of the following lines to the same file:

```
#define APPLIANCE_STATISTICS_SERVER
#define APPLIANCE_STATISTICS_CLIENT
```

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

Maximum Log Size

Add this line to configure the maximum size n, in bytes, of a data log:

```
#define CLD_APPLIANCE_STATISTICS_ATTR_LOG_MAX_SIZE n
```

The default value is 70 bytes, which is the upper limit on this value, and n must therefore not be greater than 70.

The same value must be defined on the cluster server and client.

Maximum Log Queue Length

Add this line to configure the maximum number of logs n in a log queue:

```
#define CLD_APPLIANCE_STATISTICS_ATTR_LOG_QUEUE_MAX_SIZE n
```

The default value is 15, which is the upper limit on this value, and n must therefore not be greater than 15.

The same value must be defined on the cluster server and client.

Enable Insertion of UTC Time

Add this line to enable the application to insert UTC time data into logs:

```
#define CLD_APPLIANCE_STATISTICS_ATTR_LOG_QUEUE_MAX_SIZE n
```

Part III: General Reference Information

12. HA Core Functions

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

Function	Page
eHA_Initialise	234
eHA_Update100mS	235
eHA_RegisterOnOffSwitchEndPoint	236
eHA_RegisterOnOffOutputEndPoint	238
eHA_RegisterRemoteControlEndPoint	240
eHA_RegisterDoorLockEndPoint	242
eHA_RegisterDoorLockControllerEndPoint	244
eHA_RegisterSimpleSensorEndPoint	246
eHA_RegisterOnOffLightEndPoint	248
eHA_RegisterDimmableLightEndPoint	250
eHA_RegisterColourDimmableLightEndPoint	252
eHA_RegisterOnOffLightSwitchEndPoint	254
eHA_RegisterDimmerSwitchEndPoint	256
eHA_RegisterColourDimmerSwitchEndPoint	258
eHA_RegisterLightSensorEndPoint	260
eHA_RegisterOccupancySensorEndPoint	262

Note 1: For guidance on using these functions in your application code, refer to Chapter 4.

Note 2: The return codes for these functions are described in the *ZCL User Guide (JN-UG-3077)*.

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

eHA_Initialise

teZCL_Status eHA_Initialise(
tfpZCL_ZCLCallBackFunction *cbCallBack*,
PDUM_thAPdu *hAPdu*);

Description

This function initialises the ZCL and HA 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

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

Returns

E_ZCL_SUCCESS

E_ZCL_ERR_HEAP_FAIL

E_ZCL_ERR_PARAMETER_NULL

eHA_Update100mS

```
teZCL_Status eHA_Update100mS(void);
```

Description

This function is used to service all the timing needs of the clusters used by the HA 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

eHA_RegisterOnOffSwitchEndPoint

teZCL_Status eHA_RegisterOnOffSwitchEndPoint(
uint8 *u8EndPointIdentifier*,
tfpZCL_ZCLCallBackFunction *cbCallBack*,
tsHA_OnOffSwitchDevice **psDeviceInfo***);**

Description

This function is used to register an endpoint which will support an On/Off Switch device. The function must be called after the **eHA_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). HA endpoints are normally numbered consecutively starting at 1. The specified number must be less than or equal to the value of HA_NUMBER_OF_ENDPOINTS defined in the **zcl_options.h** file, which represents the highest endpoint number used for HA.

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 `tsHA_OnOffSwitchDevice` structure (see Section 13.1.1) which will be used to store all variables relating to the On/Off Switch device associated with the endpoint. The `sEndPoint` and `sClusterInstance` fields of this structure are set by this function and must not be directly written to by the application.

The function may be called multiple times if more than one endpoint is being used - for example, if more than one On/Off Switch device is housed in the same hardware, sharing the same JN5168 module.

Parameters

u8EndPointIdentifier	Endpoint that is to be associated with the registered structure and callback function
cbCallBack	Pointer to the function that the HA library will use to indicate events to the application for this endpoint
psDeviceInfo	Pointer to the structure that will act as storage for all variables related to the device being registered on this endpoint (see Section 13.1.1). The `sEndPoint` and `sClusterInstance` fields are set by this register function for internal use and must not be written to by the application

Returns

E_ZCL_SUCCESS

E_ZCL_FAIL

E_ZCL_ERR_PARAMETER_NULL

E_ZCL_ERR_PARAMETER_RANGE

E_ZCL_ERR_EP_RANGE

E_ZCL_ERR_CLUSTER_0

E_ZCL_ERR_CALLBACK_NULL

eHA_RegisterOnOffOutputEndPoint

```
teZCL_Status eHA_RegisterOnOffOutputEndPoint(
        uint8 u8EndPointIdentifier,
        tfpZCL_ZCLCallBackFunction cbCallBack,
        tsHA_OnOffOutputDevice *psDeviceInfo);
```

Description

This function is used to register an endpoint which will support an On/Off Output device. The function must be called after the **eHA_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). HA endpoints are normally numbered consecutively starting at 1. The specified number must be less than or equal to the value of HA_NUMBER_OF_ENDPOINTS defined in the **zcl_options.h** file, which represents the highest endpoint number used for HA.

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 `tsHA_OnOffOutputDevice` structure (see Section 13.1.2) which will be used to store all variables relating to the On/Off Output 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 Output device is housed in the same hardware, sharing the same JN5168 module.

Parameters

u8EndPointIdentifier	Endpoint that is to be associated with the registered structure and callback function
cbCallBack	Pointer to the function that the HA library will use to indicate events to the application for this endpoint
psDeviceInfo	Pointer to the structure that will act as storage for all variables related to the device being registered on this endpoint (see Section 13.1.2). The `sEndPoint` and `sClusterInstance` fields are set by this register function for internal use and must not be written to by the application

Returns

E_ZCL_SUCCESS
E_ZCL_FAIL
E_ZCL_ERR_PARAMETER_NULL
E_ZCL_ERR_PARAMETER_RANGE
E_ZCL_ERR_EP_RANGE
E_ZCL_ERR_CLUSTER_0
E_ZCL_ERR_CALLBACK_NULL

eHA_RegisterRemoteControlEndPoint

teZCL_Status eHA_RegisterRemoteControlEndPoint(
uint8 *u8EndPointIdentifier*,
tfpZCL_ZCLCallBackFunction *cbCallBack*,
tsHA_RemoteControlDevice **psDeviceInfo*);

Description

This function is used to register an endpoint which will support a Remote Control device. The function must be called after the **eHA_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). HA endpoints are normally numbered consecutively starting at 1. The specified number must be less than or equal to the value of HA_NUMBER_OF_ENDPOINTS defined in the **zcl_options.h** file, which represents the highest endpoint number used for HA.

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 `tsHA_RemoteControlDevice` structure (see Section 13.1.3) which will be used to store all variables relating to the Remote Control 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 Remote Control device is housed in the same hardware, sharing the same JN5168 module.

Parameters

u8EndPointIdentifier	Endpoint that is to be associated with the registered structure and callback function
cbCallBack	Pointer to the function that the HA library will use to indicate events to the application for this endpoint
psDeviceInfo	Pointer to the structure that will act as storage for all variables related to the device being registered on this endpoint (see Section 13.1.3). The `sEndPoint` and `sClusterInstance` fields are set by this register function for internal use and must not be written to by the application

Returns

E_ZCL_SUCCESS
E_ZCL_FAIL
E_ZCL_ERR_PARAMETER_NULL
E_ZCL_ERR_PARAMETER_RANGE
E_ZCL_ERR_EP_RANGE
E_ZCL_ERR_CLUSTER_0
E_ZCL_ERR_CALLBACK_NULL

eHA_RegisterDoorLockEndPoint

teZCL_Status eHA_RegisterDoorLockEndPoint(
uint8 *u8EndPointIdentifier*,
tfpZCL_ZCLCallBackFunction *cbCallBack*,
tsHA_DoorLockDevice **psDeviceInfo*);

Description

This function is used to register an endpoint which will support a Door Lock device. The function must be called after the **eHA_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). HA endpoints are normally numbered consecutively starting at 1. The specified number must be less than or equal to the value of HA_NUMBER_OF_ENDPOINTS defined in the **zcl_options.h** file, which represents the highest endpoint number used for HA.

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 `tsHA_DoorLockDevice` structure (see Section 13.1.4) which will be used to store all variables relating to the Door Lock 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 Door Lock device is housed in the same hardware, sharing the same JN5168 module.

Parameters

u8EndPointIdentifier	Endpoint that is to be associated with the registered structure and callback function
cbCallBack	Pointer to the function that the HA library will use to indicate events to the application for this endpoint
psDeviceInfo	Pointer to the structure that will act as storage for all variables related to the device being registered on this endpoint (see Section 13.1.4). The `sEndPoint` and `sClusterInstance` fields are set by this register function for internal use and must not be written to by the application

Returns

E_ZCL_SUCCESS
E_ZCL_FAIL
E_ZCL_ERR_PARAMETER_NULL
E_ZCL_ERR_PARAMETER_RANGE
E_ZCL_ERR_EP_RANGE
E_ZCL_ERR_CLUSTER_0
E_ZCL_ERR_CALLBACK_NULL

eHA_RegisterDoorLockControllerEndPoint

```
teZCL_Status eHA_RegisterDoorLockControllerEndPoint(
        uint8 u8EndPointIdentifier,
        tfpZCL_ZCLCallBackFunction cbCallBack,
        tsHA_DoorLockControllerDevice *psDeviceInfo);
```

Description

This function is used to register an endpoint which will support a Door Lock Controller device. The function must be called after the **eHA_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). HA endpoints are normally numbered consecutively starting at 1. The specified number must be less than or equal to the value of HA_NUMBER_OF_ENDPOINTS defined in the **zcl_options.h** file, which represents the highest endpoint number used for HA.

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 `tsHA_DoorLockControllerDevice` structure (see Section 13.1.5) which will be used to store all variables relating to the Door Lock 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 Door Lock Controller device is housed in the same hardware, sharing the same JN5168 module.

Parameters

u8EndPointIdentifier	Endpoint that is to be associated with the registered structure and callback function
cbCallBack	Pointer to the function that the HA library will use to indicate events to the application for this endpoint
psDeviceInfo	Pointer to the structure that will act as storage for all variables related to the device being registered on this endpoint (see Section 13.1.5). The `sEndPoint` and `sClusterInstance` fields are set by this register function for internal use and must not be written to by the application

Returns

E_ZCL_SUCCESS
E_ZCL_FAIL
E_ZCL_ERR_PARAMETER_NULL
E_ZCL_ERR_PARAMETER_RANGE
E_ZCL_ERR_EP_RANGE
E_ZCL_ERR_CLUSTER_0
E_ZCL_ERR_CALLBACK_NULL

eHA_RegisterSimpleSensorEndPoint

teZCL_Status eHA_RegisterSimpleSensorEndPoint(
uint8 *u8EndPointIdentifier*,
tfpZCL_ZCLCallBackFunction *cbCallBack*,
tsHA_SimpleSensorDevice ****psDeviceInfo*);**

Description

This function is used to register an endpoint which will support a Simple Sensor device. The function must be called after the **eHA_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). HA endpoints are normally numbered consecutively starting at 1. The specified number must be less than or equal to the value of HA_NUMBER_OF_ENDPOINTS defined in the **zcl_options.h** file, which represents the highest endpoint number used for HA.

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 `tsHA_SimpleSensorDevice` structure (see Section 13.1.6) which will be used to store all variables relating to the Simple 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 Simple Sensor device is housed in the same hardware, sharing the same JN5168 module.

Parameters

u8EndPointIdentifier	Endpoint that is to be associated with the registered structure and callback function
cbCallBack	Pointer to the function that the HA library will use to indicate events to the application for this endpoint
psDeviceInfo	Pointer to the structure that will act as storage for all variables related to the device being registered on this endpoint (see Section 13.1.6). The `sEndPoint` and `sClusterInstance` fields are set by this register function for internal use and must not be written to by the application

Returns

E_ZCL_SUCCESS
E_ZCL_FAIL
E_ZCL_ERR_PARAMETER_NULL
E_ZCL_ERR_PARAMETER_RANGE
E_ZCL_ERR_EP_RANGE
E_ZCL_ERR_CLUSTER_0
E_ZCL_ERR_CALLBACK_NULL

eHA_RegisterOnOffLightEndPoint

teZCL_Status eHA_RegisterOnOffLightEndPoint(
uint8 *u8EndPointIdentifier*,
tfpZCL_ZCLCallBackFunction *cbCallBack*,
tsHA_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 **eHA_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). HA endpoints are normally numbered consecutively starting at 1. The specified number must be less than or equal to the value of HA_NUMBER_OF_ENDPOINTS defined in the **zcl_options.h** file, which represents the highest endpoint number used for HA.

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 `tsHA_OnOffLightDevice` structure (see Section 13.2.1) 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 JN5168 module.

Parameters

u8EndPointIdentifier	Endpoint that is to be associated with the registered structure and callback function
cbCallBack	Pointer to the function that the HA library will use to indicate events to the application for this endpoint
psDeviceInfo	Pointer to the structure that will act as storage for all variables related to the device being registered on this endpoint (see Section 13.2.1). The `sEndPoint` and `sClusterInstance` fields are set by this register function for internal use and must not be written to by the application

Returns

E_ZCL_SUCCESS
E_ZCL_FAIL
E_ZCL_ERR_PARAMETER_NULL
E_ZCL_ERR_PARAMETER_RANGE
E_ZCL_ERR_EP_RANGE
E_ZCL_ERR_CLUSTER_0
E_ZCL_ERR_CALLBACK_NULL

eHA_RegisterDimmableLightEndPoint

teZCL_Status eHA_RegisterDimmableLightEndPoint(
uint8 *u8EndPointIdentifier*,
tfpZCL_ZCLCallBackFunction *cbCallBack*,
tsHA_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 **eHA_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). HA endpoints are normally numbered consecutively starting at 1. The specified number must be less than or equal to the value of HA_NUMBER_OF_ENDPOINTS defined in the **zcl_options.h** file, which represents the highest endpoint number used for HA.

As part of this function call, you must specify a user-defined callback function that will be invoked when an event occurs that is associated with the endpoint (events are detailed in the *ZCL User Guide (JN-UG-3077)*). This callback function is defined according to the typedef:

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

You must also provide a pointer to a `tsHA_DimmableLightDevice` structure (see Section 13.2.2) which will be used to store all variables relating to the Dimmable Light device associated with the endpoint. The `sEndPoint` and `sClusterInstance` fields of this structure are set by this function and must not be directly written to by the application.

The function may be called multiple times if more than one endpoint is being used - for example, if more than one Dimmable Light device is housed in the same hardware, sharing the same JN5168 module.

Parameters

u8EndPointIdentifier	Endpoint that is to be associated with the registered structure and callback function
cbCallBack	Pointer to the function that the HA library will use to indicate events to the application for this endpoint
psDeviceInfo	Pointer to the structure that will act as storage for all variables related to the device being registered on this endpoint (see Section 13.2.2). The `sEndPoint` and `sClusterInstance` fields are set by this register function for internal use and must not be written to by the application

Returns

E_ZCL_SUCCESS
E_ZCL_FAIL
E_ZCL_ERR_PARAMETER_NULL
E_ZCL_ERR_PARAMETER_RANGE
E_ZCL_ERR_EP_RANGE
E_ZCL_ERR_CLUSTER_0
E_ZCL_ERR_CALLBACK_NULL

eHA_RegisterColourDimmableLightEndPoint

```
teZCL_Status eHA_RegisterColourDimmableLightEndPoint(
        uint8 u8EndPointIdentifier,
        tfpZCL_ZCLCallBackFunction cbCallBack,
        tsHA_ColourDimmableLightDevice *psDeviceInfo);
```

Description

This function is used to register an endpoint which will support a Colour Dimmable Light device. The function must be called after the **eHA_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). HA endpoints are normally numbered consecutively starting at 1. The specified number must be less than or equal to the value of HA_NUMBER_OF_ENDPOINTS defined in the **zcl_options.h** file, which represents the highest endpoint number used for HA.

As part of this function call, you must specify a user-defined callback function that will be invoked when an event occurs that is associated with the endpoint (events are detailed in the *ZCL User Guide (JN-UG-3077)*). This callback function is defined according to the typedef:

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

You must also provide a pointer to a `tsHA_ColourDimmableLightDevice` structure (see Section 13.2.3) which will be used to store all variables relating to the Colour Dimmable Light device associated with the endpoint. The `sEndPoint` and `sClusterInstance` fields of this structure are set by this function and must not be directly written to by the application.

The function may be called multiple times if more than one endpoint is being used - for example, if more than one Colour Dimmable Light device is housed in the same hardware, sharing the same JN5168 module.

Parameters

u8EndPointIdentifier	Endpoint that is to be associated with the registered structure and callback function
cbCallBack	Pointer to the function that the HA library will use to indicate events to the application for this endpoint
psDeviceInfo	Pointer to the structure that will act as storage for all variables related to the device being registered on this endpoint (see Section 13.2.3). The `sEndPoint` and `sClusterInstance` fields are set by this register function for internal use and must not be written to by the application

Returns

E_ZCL_SUCCESS
E_ZCL_FAIL
E_ZCL_ERR_PARAMETER_NULL
E_ZCL_ERR_PARAMETER_RANGE
E_ZCL_ERR_EP_RANGE
E_ZCL_ERR_CLUSTER_0
E_ZCL_ERR_CALLBACK_NULL

eHA_RegisterOnOffLightSwitchEndPoint

```
teZCL_Status eHA_RegisterOnOffLightSwitchEndPoint(
            uint8 u8EndPointIdentifier,
            tfpZCL_ZCLCallBackFunction cbCallBack,
            tsHA_OnOffLightSwitchDevice *psDeviceInfo);
```

Description

This function is used to register an endpoint which will support an On/Off Light Switch device. The function must be called after the **eHA_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). HA endpoints are normally numbered consecutively starting at 1. The specified number must be less than or equal to the value of HA_NUMBER_OF_ENDPOINTS defined in the **zcl_options.h** file, which represents the highest endpoint number used for HA.

As part of this function call, you must specify a user-defined callback function that will be invoked when an event occurs that is associated with the endpoint (events are detailed in the *ZCL User Guide (JN-UG-3077)*). This callback function is defined according to the typedef:

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

You must also provide a pointer to a `tsHA_OnOffLightSwitchDevice` structure (see Section 13.2.4) which will be used to store all variables relating to the On/Off Light Switch device associated with the endpoint. The `sEndPoint` and `sClusterInstance` fields of this structure are set by this function and must not be directly written to by the application.

The function may be called multiple times if more than one endpoint is being used - for example, if more than one On/Off Light Switch device is housed in the same hardware, sharing the same JN5168 module.

Parameters

u8EndPointIdentifier	Endpoint that is to be associated with the registered structure and callback function
cbCallBack	Pointer to the function that the HA library will use to indicate events to the application for this endpoint
psDeviceInfo	Pointer to the structure that will act as storage for all variables related to the device being registered on this endpoint (see Section 13.2.4). The `sEndPoint` and `sClusterInstance` fields are set by this register function for internal use and must not be written to by the application

Returns

E_ZCL_SUCCESS
E_ZCL_FAIL
E_ZCL_ERR_PARAMETER_NULL
E_ZCL_ERR_PARAMETER_RANGE
E_ZCL_ERR_EP_RANGE
E_ZCL_ERR_CLUSTER_0
E_ZCL_ERR_CALLBACK_NULL

eHA_RegisterDimmerSwitchEndPoint

```
teZCL_Status eHA_RegisterDimmerSwitchEndPoint(
        uint8 u8EndPointIdentifier,
        tfpZCL_ZCLCallBackFunction cbCallBack,
        tsHA_DimmerSwitchDevice *psDeviceInfo);
```

Description

This function is used to register an endpoint which will support a Dimmer Switch device. The function must be called after the **eHA_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). HA endpoints are normally numbered consecutively starting at 1. The specified number must be less than or equal to the value of HA_NUMBER_OF_ENDPOINTS defined in the **zcl_options.h** file, which represents the highest endpoint number used for HA.

As part of this function call, you must specify a user-defined callback function that will be invoked when an event occurs that is associated with the endpoint (events are detailed in the *ZCL User Guide (JN-UG-3077)*). This callback function is defined according to the typedef:

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

You must also provide a pointer to a `tsHA_DimmerSwitchDevice` structure (see Section 13.2.5) which will be used to store all variables relating to the Dimmer Switch device associated with the endpoint. The `sEndPoint` and `sClusterInstance` fields of this structure are set by this function and must not be directly written to by the application.

The function may be called multiple times if more than one endpoint is being used - for example, if more than one Dimmer Switch device is housed in the same hardware, sharing the same JN5168 module.

Parameters

u8EndPointIdentifier	Endpoint that is to be associated with the registered structure and callback function
cbCallBack	Pointer to the function that the HA library will use to indicate events to the application for this endpoint
psDeviceInfo	Pointer to the structure that will act as storage for all variables related to the device being registered on this endpoint (see Section 13.2.5). The `sEndPoint` and `sClusterInstance` fields are set by this register function for internal use and must not be written to by the application

Returns

E_ZCL_SUCCESS
E_ZCL_FAIL
E_ZCL_ERR_PARAMETER_NULL
E_ZCL_ERR_PARAMETER_RANGE
E_ZCL_ERR_EP_RANGE
E_ZCL_ERR_CLUSTER_0
E_ZCL_ERR_CALLBACK_NULL

eHA_RegisterColourDimmerSwitchEndPoint

teZCL_Status eHA_RegisterColourDimmerSwitchEndPoint(
uint8 *u8EndPointIdentifier*,
tfpZCL_ZCLCallBackFunction *cbCallBack*,
tsHA_DimmerSwitchDevice **psDeviceInfo*);

Description

This function is used to register an endpoint which will support a Colour Dimmer Switch device. The function must be called after the **eHA_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). HA endpoints are normally numbered consecutively starting at 1. The specified number must be less than or equal to the value of HA_NUMBER_OF_ENDPOINTS defined in the **zcl_options.h** file, which represents the highest endpoint number used for HA.

As part of this function call, you must specify a user-defined callback function that will be invoked when an event occurs that is associated with the endpoint (events are detailed in the *ZCL User Guide (JN-UG-3077)*). This callback function is defined according to the typedef:

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

You must also provide a pointer to a `tsHA_ColourDimmerSwitchDevice` structure (see Section 13.2.6) which will be used to store all variables relating to the Colour Dimmer Switch device associated with the endpoint. The `sEndPoint` and `sClusterInstance` fields of this structure are set by this function and must not be directly written to by the application.

The function may be called multiple times if more than one endpoint is being used - for example, if more than one Colour Dimmer Switch device is housed in the same hardware, sharing the same JN5168 module.

Parameters

u8EndPointIdentifier	Endpoint that is to be associated with the registered structure and callback function
cbCallBack	Pointer to the function that the HA library will use to indicate events to the application for this endpoint
psDeviceInfo	Pointer to the structure that will act as storage for all variables related to the device being registered on this endpoint (see Section 13.2.6). The `sEndPoint` and `sClusterInstance` fields are set by this register function for internal use and must not be written to by the application

Returns

E_ZCL_SUCCESS
E_ZCL_FAIL
E_ZCL_ERR_PARAMETER_NULL
E_ZCL_ERR_PARAMETER_RANGE
E_ZCL_ERR_EP_RANGE
E_ZCL_ERR_CLUSTER_0
E_ZCL_ERR_CALLBACK_NULL

eHA_RegisterLightSensorEndPoint

teZCL_Status eHA_RegisterLightSensorEndPoint(
    uint8 *u8EndPointIdentifier*,
    tfpZCL_ZCLCallBackFunction *cbCallBack*,
    tsHA_LightSensorDevice ****psDeviceInfo*);**

Description

This function is used to register an endpoint which will support a Light Sensor device. The function must be called after the **eHA_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). HA endpoints are normally numbered consecutively starting at 1. The specified number must be less than or equal to the value of HA_NUMBER_OF_ENDPOINTS defined in the **zcl_options.h** file, which represents the highest endpoint number used for HA.

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 `tsHA_LightSensorDevice` structure (see Section 13.2.7) which will be used to store all variables relating to the Light Sensor device associated with the endpoint. The `sEndPoint` and `sClusterInstance` fields of this structure are set by this function and must not be directly written to by the application.

The function may be called multiple times if more than one endpoint is being used - for example, if more than one Light Sensor device is housed in the same hardware, sharing the same JN5168 module.

Parameters

u8EndPointIdentifier	Endpoint that is to be associated with the registered structure and callback function
cbCallBack	Pointer to the function that the HA library will use to indicate events to the application for this endpoint
psDeviceInfo	Pointer to the structure that will act as storage for all variables related to the device being registered on this endpoint (see Section 13.2.7). The `sEndPoint` and `sClusterInstance` fields are set by this register function for internal use and must not be written to by the application

Returns

E_ZCL_SUCCESS
E_ZCL_FAIL
E_ZCL_ERR_PARAMETER_NULL
E_ZCL_ERR_PARAMETER_RANGE
E_ZCL_ERR_EP_RANGE
E_ZCL_ERR_CLUSTER_0
E_ZCL_ERR_CALLBACK_NULL

eHA_RegisterOccupancySensorEndPoint

teZCL_Status eHA_RegisterOccupancySensorEndPoint(
uint8 *u8EndPointIdentifier*,
tfpZCL_ZCLCallBackFunction *cbCallBack*,
tsHA_OccupancySensorDevice ****psDeviceInfo*);**

Description

This function is used to register an endpoint which will support an Occupancy Sensor device. The function must be called after the **eHA_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). HA endpoints are normally numbered consecutively starting at 1. The specified number must be less than or equal to the value of HA_NUMBER_OF_ENDPOINTS defined in the **zcl_options.h** file, which represents the highest endpoint number used for HA.

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 `tsHA_OccupancySensorDevice` structure (see Section 13.2.8) which will be used to store all variables relating to the Light Sensor device associated with the endpoint. The `sEndPoint` and `sClusterInstance` fields of this structure are set by this function and must not be directly written to by the application.

The function may be called multiple times if more than one endpoint is being used - for example, if more than one Occupancy Sensor device is housed in the same hardware, sharing the same JN5168 module.

Parameters

u8EndPointIdentifier	Endpoint that is to be associated with the registered structure and callback function
cbCallBack	Pointer to the function that the HA library will use to indicate events to the application for this endpoint
psDeviceInfo	Pointer to the structure that will act as storage for all variables related to the device being registered on this endpoint (see Section 13.2.8). The `sEndPoint` and `sClusterInstance` fields are set by this register function for internal use and must not be written to by the application

Returns

E_ZCL_SUCCESS
E_ZCL_FAIL
E_ZCL_ERR_PARAMETER_NULL
E_ZCL_ERR_PARAMETER_RANGE
E_ZCL_ERR_EP_RANGE
E_ZCL_ERR_CLUSTER_0
E_ZCL_ERR_CALLBACK_NULL

13. HA Device Structures

This chapter presents the shared device structures for the HA devices supported by the HA API. The supported HA devices are introduced in Chapter 2.

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.

13.1 Generic Devices

The structures for the following Generic Devices are presented in this section:

- On/Off Switch (`tsHA_OnOffSwitchDevice`) - see Section 13.1.1
- On/Off Output (`tsHA_OnOffOutputDevice`) - see Section 13.1.2
- Remote Control (`tsHA_RemoteControlDevice`) - see Section 13.1.4
- Door Lock (`tsHA_DoorLockDevice`) - see Section 13.1.4
- Door Lock Controller (`tsHA_DoorLockControllerDevice`) - see Section 13.1.5
- Simple Sensor (`tsHA_SimpleSensorDevice`) - see Section 13.1.6

13.1.1 tsHA_OnOffSwitchDevice

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

```
typedef struct
{
    tsZCL_EndPointDefinition sEndPoint;

    /* Cluster instances */
    tsHA_OnOffSwitchDeviceClusterInstances 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_OOSC) && (defined OOSC_SERVER)
    /* On/Off Switch Configuration Cluster - Server */
    tsCLD_OnOff sOOSCServerCluster;
#endif

    /* Optional server clusters */
#if (defined CLD_POWER_CONFIGURATION) && (defined
POWER_CONFIGURATION_SERVER)
    /* Power Configuration Cluster - Server */
    tsCLD_PowerConfiguration sPowerConfigServerCluster;
#endif

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

#if (defined CLD_ALARMS) && (defined ALARMS_SERVER)
    /* Alarms Cluster - Server */
    tsCLD_Alarms sAlarmsServerCluster;
    tsCLD_AlarmsCustomDataStructure sAlarmsServerCustomDataStructure;
#endif

    /* Mandatory client clusters */
#if (defined CLD_ONOFF) && (defined ONOFF_CLIENT)
    /* On/Off Cluster - Client */
    tsCLD_OnOff sOnOffClientCluster;
#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_IDENTIFY) && (defined IDENTIFY_CLIENT)
    /* Identify Cluster - Client */
    tsCLD_Identify sIdentifyClientCluster;
    tsCLD_IdentifyCustomDataStructure
sIdentifyClientCustomDataStructure;
#endif

} tsHA_OnOffSwitchDevice;
```

13.1.2 tsHA_OnOffOutputDevice

The following `tsHA_OnOffOutputDevice` structure is the shared structure for an On/Off Output device:

```
typedef struct
{
    tsZCL_EndPointDefinition sEndPoint;

    /* Cluster instances */
    tsHA_OnOffOutputDeviceClusterInstances sClusterInstance;

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

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

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

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

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

    /* Optional server clusters */
#if (defined CLD_POWER_CONFIGURATION) && (defined 
POWER_CONFIGURATION_SERVER)
    /* Power Configuration Cluster - Server */
    tsCLD_PowerConfiguration sPowerConfigServerCluster;
#endif

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

#if (defined CLD_ALARMS) && (defined ALARMS_SERVER)
    /* Alarms Cluster - Server */
    tsCLD_Alarms sAlarmsServerCluster;
    tsCLD_AlarmsCustomDataStructure 
sAlarmsServerCustomDataStructure;
```

```
#endif

} tsHA_OnOffOutputDevice;
```

13.1.3 tsHA_RemoteControlDevice

The following `tsHA_RemoteControlDevice` structure is the shared structure for a Remote Control device:

```
typedef struct
{
    tsZCL_EndPointDefinition sEndPoint;

    /* Cluster instances */
    tsHA_RemoteControlDeviceClusterInstances sClusterInstance;

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

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

    /* Optional server clusters */
#if (defined CLD_POWER_CONFIGURATION) && (defined
POWER_CONFIGURATION_SERVER)
    /* Power Configuration Cluster - Server */
    tsCLD_PowerConfiguration sPowerConfigServerCluster;
#endif

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

#if (defined CLD_ALARMS) && (defined ALARMS_SERVER)
    /* Alarms Cluster - Server */
    tsCLD_Alarms sAlarmsServerCluster;
    tsCLD_AlarmsCustomDataStructure 
sAlarmsServerCustomDataStructure;
#endif

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

#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

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

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

#if (defined CLD_OOSC) && (defined OOSC_CLIENT)
    /* On/Off Switch Configuration Cluster - Client */
    tsCLD_OnOff sOOSCClientCluster;
#endif

#if (defined CLD_TEMPERATURE_MEASUREMENT) && (defined
TEMPERATURE_MEASUREMENT_CLIENT)
    /* Temperature Measurement Cluster - Client */
    tsCLD_TemperatureMeasurement
sTemperatureMeasurementClientCluster;
#endif

#if (defined CLD_ILLUMINANCE_LEVEL_SENSING) && (defined
ILLUMINANCE_LEVEL_SENSING_CLIENT)
    /* Illuminance Level Sensing Cluster - Client */
    tsCLD_IlluminanceLevelSensing
sIlluminanceLevelSensingClientCluster;
#endif

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

} tsHA_RemoteControlDevice;
```

13.1.4 tsHA_DoorLockDevice

The following `tsHA_DoorLockDevice` structure is the shared structure for a Door Lock device:

```
typedef struct
{
    tsZCL_EndPointDefinition sEndPoint;

    /* Cluster instances */
    tsHA_DoorLockDeviceClusterInstances sClusterInstance;

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

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

#if (defined CLD_DOOR_LOCK) && (defined DOOR_LOCK_SERVER)
    /* door lock Cluster - Server */
    tsCLD_DoorLock sDoorLockServerCluster;

#endif

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

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

```
    /* Optional server clusters */
#if (defined CLD_POWER_CONFIGURATION) && (defined
POWER_CONFIGURATION_SERVER)
    /* Power Configuration Cluster - Server */
    tsCLD_PowerConfiguration sPowerConfigServerCluster;
 #endif

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

#if (defined CLD_ALARMS) && (defined ALARMS_SERVER)
    /* Alarms Cluster - Server */
    tsCLD_Alarms sAlarmsServerCluster;
    tsCLD_AlarmsCustomDataStructure
sAlarmsServerCustomDataStructure;
#endif

} tsHA_DoorLockDevice;
```

13.1.5 tsHA_DoorLockControllerDevice

The following `tsHA_DoorLockControllerDevice` structure is the shared structure for a Door Lock Controller device:

```
typedef struct
{
    tsZCL_EndPointDefinition sEndPoint;

    /* Cluster instances */
   tsHA_DoorLockControllerDeviceClusterInstances sClusterInstance;

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

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

    /* Optional server clusters */
#if (defined CLD_POWER_CONFIGURATION) && (defined
POWER_CONFIGURATION_SERVER)
    /* Power Configuration Cluster - Server */
    tsCLD_PowerConfiguration sPowerConfigServerCluster;
#endif

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

#if (defined CLD_ALARMS) && (defined ALARMS_SERVER)
    /* Alarms Cluster - Server */
    tsCLD_Alarms sAlarmsServerCluster;
    tsCLD_AlarmsCustomDataStructure
sAlarmsServerCustomDataStructure;
#endif

    /* Mandatory client clusters */
#if (defined CLD_DOOR_LOCK) && (defined DOOR_LOCK_CLIENT)
    /* Door Lock Cluster - Client */
    tsCLD_DoorLock sDoorLockClientCluster;
#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_IDENTIFY) && (defined IDENTIFY_CLIENT)
    /* Identify Cluster - Client */
    tsCLD_Identify sIdentifyClientCluster;
    tsCLD_IdentifyCustomDataStructure 
sIdentifyClientCustomDataStructure;
#endif

} tsHA_DoorLockControllerDevice;
```

13.1.6 tsHA_SimpleSensorDevice

The following `tsHA_SimpleSensorDevice` structure is the shared structure for a Simple Sensor device:

```
typedef struct
{
    tsZCL_EndPointDefinition sEndPoint;

    /* Cluster instances */
    tsHA_SimpleSensorDeviceClusterInstances sClusterInstance;

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

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

#if (defined CLD_BINARY_INPUT_BASIC) && (defined 
BINARY_INPUT_BASIC_SERVER)
    /* Binary Input Basic Cluster - Server */
    tsCLD_BinaryInputBasic sBinaryInputBasicServerCluster;
#endif

    /* Optional server clusters */
#if (defined CLD_POWER_CONFIGURATION) && (defined 
POWER_CONFIGURATION_SERVER)
    /* Power Configuration Cluster - Server */
    tsCLD_PowerConfiguration sPowerConfigServerCluster;
 #endif

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

#if (defined CLD_ALARMS) && (defined ALARMS_SERVER)
    /* Alarms Cluster - Server */
    tsCLD_Alarms sAlarmsServerCluster;
    tsCLD_AlarmsCustomDataStructure 
sAlarmsServerCustomDataStructure;
#endif

} tsHA_SimpleSensorDevice;
```

13.2 Lighting Devices

The structures for the following Lighting Devices are presented in this section:

- On/Off Light (`tsHA_OnOffLightDevice`) - see Section 13.2.1
- Dimmable Light (`tsHA_DimmableLightDevice`) - see Section 13.2.2
- Colour Dimmable Light (`tsHA_ColourDimmableLightDevice`) - see Section 13.2.3
- On/Off Light Switch (`tsHA_DimmableLightDevice`) - see Section 13.2.4
- Dimmer Switch (`tsHA_DimmerSwitchDevice`) - see Section 13.2.5
- Colour Dimmer Switch (`tsHA_ColourDimmerSwitchDevice`) - see Section 13.2.6
- Light Sensor (`tsHA_LightSensorDevice`) - see Section 13.2.7
- Occupancy Sensor (`tsHA_OccupancySensorDevice`) - see Section 13.2.8

13.2.1 tsHA_OnOffLightDevice

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

```
typedef struct
{
    tsZCL_EndPointDefinition sEndPoint;

    /* Cluster instances */
    tsHA_OnOffLightDeviceClusterInstances sClusterInstance;

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

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

#if (defined CLD_ONOFF) && (defined ONOFF_SERVER)
```

```
    /* On/Off Cluster - Server */
    tsCLD_OnOff sOnOffServerCluster;
    //tsCLD_AS_OnOff sOnOffServerClusterAttributeStatus;
    tsCLD_OnOffCustomDataStructure
sOnOffServerCustomDataStructure;
#endif

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

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

    /* Optional server clusters */
#if (defined CLD_POWER_CONFIGURATION) && (defined
POWER_CONFIGURATION_SERVER)
    /* Power Configuration Cluster - Server */
    tsCLD_PowerConfiguration sPowerConfigServerCluster;
    //tsCLD_AS_PowerConfiguration
sPowerConfigServerClusterAttributeStatus;
#endif

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

#if (defined CLD_ALARMS) && (defined ALARMS_SERVER)
    /* Alarms Cluster - Server */
    tsCLD_Alarms sAlarmsServerCluster;
    //tsCLD_AS_Alarms sAlarmsServerClusterAttributeStatus;
    tsCLD_AlarmsCustomDataStructure 
sAlarmsServerCustomDataStructure;
#endif

    /* Optional client clusters */
#if (defined CLD_OCCUPANCY_SENSING) && (defined 
OCCUPANCY_SENSING_CLIENT)
    /* Occupancy Sensing Cluster - Client */
    tsCLD_OccupancySensing sOccupancySensingClientCluster;
    //tsCLD_AS_OccupancySensing 
sOccupancySensingClientClusterAttributeStatus;
#endif

} tsHA_OnOffLightDevice;
```

13.2.2 tsHA_DimmableLightDevice

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

```
typedef struct
{
    tsZCL_EndPointDefinition sEndPoint;

    /* Cluster instances */
    tsHA_DimmableLightDeviceClusterInstances sClusterInstance;

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

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

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

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

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

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

    /* Optional server clusters */
#if (defined CLD_POWER_CONFIGURATION) && (defined
POWER_CONFIGURATION_SERVER)
    /* Power Configuration Cluster - Server */
    tsCLD_PowerConfiguration sPowerConfigServerCluster;
    tsCLD_AS_PowerConfiguration
sPowerConfigServerClusterAttributeStatus;
#endif

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

```
    tsCLD_AS_DeviceTemperatureConfiguration 
sDeviceTemperatureConfigurationServerClusterAttributeStatus;
#endif

#if (defined CLD_ALARMS) && (defined ALARMS_SERVER)
    /* Alarms Cluster - Server */
    tsCLD_Alarms sAlarmsServerCluster;
    tsCLD_AS_Alarms sAlarmsServerClusterAttributeStatus;
    tsCLD_AlarmsCustomDataStructure 
sAlarmsServerCustomDataStructure;
#endif

    /* Optional client clusters */
#if (defined CLD_OCCUPANCY_SENSING) && (defined 
OCCUPANCY_SENSING_CLIENT)
    /* Occupancy Sensing Cluster - Client */
    tsCLD_OccupancySensing sOccupancySensingClientCluster;
    tsCLD_AS_OccupancySensing 
sOccupancySensingClientClusterAttributeStatus;
#endif

} tsHA_DimmableLightDevice;
```

13.2.3 tsHA_ColourDimmableLightDevice

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

```
typedef struct
{
    tsZCL_EndPointDefinition sEndPoint;

    /* Cluster instances */
    tsHA_ColourDimmableLightDeviceClusterInstances 
sClusterInstance;

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

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

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

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

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

#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

    /* Optional server clusters */
#if (defined CLD_POWER_CONFIGURATION) && (defined
POWER_CONFIGURATION_SERVER)
    /* Power Configuration Cluster - Server */
    tsCLD_PowerConfiguration sPowerConfigServerCluster;
```

```
#endif

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

#if (defined CLD_ALARMS) && (defined ALARMS_SERVER)
    /* Alarms Cluster - Server */
    tsCLD_Alarms sAlarmsServerCluster;
    tsCLD_AlarmsCustomDataStructure
sAlarmsServerCustomDataStructure;
#endif

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

} tsHA_ColourDimmableLightDevice;
```

13.2.4 tsHA_OnOffLightSwitchDevice

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

```
typedef struct
{
    tsZCL_EndPointDefinition sEndPoint;

    /* Cluster instances */
    tsHA_OnOffLightSwitchDeviceClusterInstances sClusterInstance;

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

#if (defined CLD_IDENTIFY) && (defined IDENTIFY_SERVER)
```

```
    /* Identify Cluster - Server */
    tsCLD_Identify sIdentifyServerCluster;
    //tsCLD_AS_Identify sIdentifyServerClusterAttributeStatus;
    tsCLD_IdentifyCustomDataStructure 
sIdentifyServerCustomDataStructure;
#endif

#if (defined CLD_OOSC) && (defined OOSC_SERVER)
    /* On/Off Switch Configuration Cluster - Server */
    tsCLD_OnOff sOOSCServerCluster;
    //tsCLD_AS_OnOff sOOSCServerClusterAttributeStatus;
#endif

    /* Optional server clusters */
#if (defined CLD_POWER_CONFIGURATION) && (defined 
POWER_CONFIGURATION_SERVER)
    /* Power Configuration Cluster - Server */
    tsCLD_PowerConfiguration sPowerConfigServerCluster;
    //tsCLD_AS_PowerConfiguration 
sPowerConfigServerClusterAttributeStatus;
#endif

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

#if (defined CLD_ALARMS) && (defined ALARMS_SERVER)
    /* Alarms Cluster - Server */
    tsCLD_Alarms sAlarmsServerCluster;
    //tsCLD_AS_Alarms sAlarmsServerClusterAttributeStatus;
    tsCLD_AlarmsCustomDataStructure 
sAlarmsServerCustomDataStructure;
#endif

    /* Mandatory client clusters */
#if (defined CLD_ONOFF) && (defined ONOFF_CLIENT)
    /* On/Off Cluster - Client */
    tsCLD_OnOff sOnOffClientCluster;
    //tsCLD_AS_OnOff sOnOffClientClusterAttributeStatus;
```

```
#endif

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

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

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

} tsHA_OnOffLightSwitchDevice;
```

13.2.5 tsHA_DimmerSwitchDevice

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

```
typedef struct
{
    tsZCL_EndPointDefinition sEndPoint;

    /* Cluster instances */
    tsHA_DimmerSwitchDeviceClusterInstances sClusterInstance;

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

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

#if (defined CLD_OOSC) && (defined OOSC_SERVER)
    /* On/Off Switch Configuration Cluster - Server */
    tsCLD_OnOff sOOSCServerCluster;
    //tsCLD_AS_OnOff sOOSCServerClusterAttributeStatus;
#endif

    /* Optional server clusters */
#if (defined CLD_POWER_CONFIGURATION) && (defined
POWER_CONFIGURATION_SERVER)
    /* Power Configuration Cluster - Server */
    tsCLD_PowerConfiguration sPowerConfigServerCluster;
    //tsCLD_AS_PowerConfiguration
sPowerConfigServerClusterAttributeStatus;
#endif

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

#if (defined CLD_ALARMS) && (defined ALARMS_SERVER)
    /* Alarms Cluster - Server */
    tsCLD_Alarms sAlarmsServerCluster;
    //tsCLD_AS_Alarms sAlarmsServerClusterAttributeStatus;
    tsCLD_AlarmsCustomDataStructure
sAlarmsServerCustomDataStructure;
#endif

    /* Mandatory client clusters */
```

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

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

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

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

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

} tsHA_DimmerSwitchDevice;
```

13.2.6 tsHA_ColourDimmerSwitchDevice

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

```
typedef struct
{
    tsZCL_EndPointDefinition sEndPoint;

    /* Cluster instances */
   tsHA_ColourDimmerSwitchDeviceClusterInstances sClusterInstance;

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

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

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

    /* Optional server clusters */
#if (defined CLD_POWER_CONFIGURATION) && (defined
POWER_CONFIGURATION_SERVER)
    /* Power Configuration Cluster - Server */
    tsCLD_PowerConfiguration sPowerConfigServerCluster;
#endif

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

#if (defined CLD_ALARMS) && (defined ALARMS_SERVER)
```

```
    /* Alarms Cluster - Server */
    tsCLD_Alarms sAlarmsServerCluster;
    tsCLD_AlarmsCustomDataStructure
sAlarmsServerCustomDataStructure;
#endif

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

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

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

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

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

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

} tsHA_ColourDimmerSwitchDevice;
```

13.2.7 tsHA_LightSensorDevice

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

```
typedef struct
{
    tsZCL_EndPointDefinition sEndPoint;

    /* Cluster instances */
    tsHA_LightSensorDeviceClusterInstances sClusterInstance;

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

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

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

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

#if (defined CLD_ALARMS) && (defined ALARMS_SERVER)
    /* Alarms Cluster - Server */
    tsCLD_Alarms sAlarmsServerCluster;
    tsCLD_AlarmsCustomDataStructure 
sAlarmsServerCustomDataStructure;
#endif

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

} tsHA_LightSensorDevice;
```

13.2.8 tsHA_OccupancySensorDevice

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

```
typedef struct
{
    tsZCL_EndPointDefinition sEndPoint;

    /* Cluster instances */
    tsHA_OccupancySensorDeviceClusterInstances sClusterInstance;

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

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

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

    /* Optional server clusters */
#if (defined CLD_POWER_CONFIGURATION) && (defined
POWER_CONFIGURATION_SERVER)
    /* Power Configuration Cluster - Server */
    tsCLD_PowerConfiguration sPowerConfigServerCluster;
#endif

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

#if (defined CLD_ALARMS) && (defined ALARMS_SERVER)
    /* Alarms Cluster - Server */
    tsCLD_Alarms sAlarmsServerCluster;
    tsCLD_AlarmsCustomDataStructure
sAlarmsServerCustomDataStructure;
#endif

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

} tsHA_OccupancySensorDevice;
```

Part IV: Appendices

A. Custom Endpoints

An HA device and its associated clusters can be registered on an endpoint using the relevant device registration function, from those listed and described in Chapter 12. However, it is also possible to set up a custom endpoint which supports selected clusters (rather than a whole HA device and all of its associated clusters). Custom endpoints are particularly useful when using multiple endpoints on a single node - for example, the first endpoint may support a complete HA device (such as a Light Sensor) while one or more custom endpoints are used to support selected clusters.

A.1 HA Devices and Endpoints

When using custom endpoints, it is important to note the difference between the following HA 'devices':

- **Physical device:** This is the physical entity which is the HA network node
- **Logical HA device:** This is a software entity which implements a specific set of HA functionality on the node, e.g. On/Off Switch device

An HA network node may contain multiple endpoints, where one endpoint is used to represent the 'physical device' and other endpoints are used to support 'logical HA devices'. The following rules apply to cluster instances on endpoints:

- All cluster instances relating to a single 'logical HA device' must reside on a single endpoint.
- The Basic cluster relates to the 'physical device' rather than a 'logical HA device' instance. There can be only one Basic cluster server for the entire node, which can be implemented in either of the following ways:
  - A single cluster instance on a dedicated 'physical device' endpoint
  - A separate cluster instance on each 'logical HA device' endpoint, but each cluster instance must use the same `tsZCL_ClusterInstance` structure (and the same attribute values)

A.2 Cluster Creation Functions

For each of the following clusters, a creation function is provided which creates an instance of the cluster on an endpoint:

- Basic: **eCLD_BasicCreateBasic()**
- Identify: **eCLD_IdentifyCreateIdentify()**
- Groups: **eCLD_GroupsCreateGroups()**
- Scenes: **eCLD_ScenesCreateScenes()**
- On/Off: **eCLD_OnOffCreateOnOff()**
- On/Off Switch Configuration: **eCLD_OOSCCreateOnOffSwitchConfig()**
- Level Control: **eCLD_LevelControlCreateLevelControl()**
- Door Lock: **eCLD_DoorLockCreateDoorLock()**
- Colour Control: **eCLD_ColourControlCreateColourControl()**
- Appliance Control: **eCLD_ApplianceControlCreateApplianceControl()**
- Appliance Identification: **eCLD_ApplianceIdentificationCreateApplianceIdentification()**
- Appliance Events and Alerts: **eCLD_ApplianceEventsAndAlertsCreateApplianceEventsAndAlerts()**
- Appliance Statistics: **eCLD_ApplianceStatisticsCreateApplianceStatistics()**

More than one of the above functions can be called for the same endpoint in order to create multiple cluster instances on the endpoint.

Note: No more than one server instance and one client instance of a given cluster can be created on a single endpoint (e.g. one Identify cluster server and one Identify cluster client, but no further Identify cluster instances).

The creation functions for clusters from the ZCL are described in the *ZCL User Guide (JN-UG-3077)*. The creation functions for the remaining HA-specific clusters are described in the chapters for the corresponding clusters in this manual.

A.3 Custom Endpoint Set-up

In order to set up a custom endpoint (supporting selected clusters), you must do the following in your application code:

1. Create a structure for the custom endpoint containing details of the cluster instances and attributes supported - see Appendix A.3.1.
2. Initialise the fields of the `tsZCL_EndPointDefinition` structure for the endpoint.

3. Call the relevant cluster creation function(s) for the cluster(s) to be supported on the endpoint - see Appendix A.2.
4. Call the ZCL function **eZCL_Register()** for the endpoint.

A.3.1 Custom Endpoint Structure

In your application code, to set up a custom endpoint you must create a structure containing details of the cluster instances and attributes to be supported on the endpoint. This structure must include the following:

- A definition of the custom endpoint through a `tsZCL_EndPointDefinition` structure - for example:

  ```
  tsZCL_EndPointDefinition sEndPoint
  ```

- A structure containing a set of `tsZCL_ClusterInstance` structures for the supported cluster instances - for example:

  ```
  typedef struct
  {
      tsZCL_ClusterInstance sBasicServer;
      tsZCL_ClusterInstance sBasicClient;
      tsZCL_ClusterInstance sIdentifyServer;
      tsZCL_ClusterInstance sOnOffCluster;
      tsZCL_ClusterInstance sDoorLockCluster;
  } tsHA_AppCustomDeviceClusterInstances
  ```

  For each cluster instance that is not shared with another endpoint, the following should be specified via the relevant `tsZCL_ClusterInstance` structure:

  - Attribute definitions, if any - for example, the `tsCLD_Basic` structure for the Basic cluster
  - Custom data structures, if any - for example, the `tsIdentify_CustomStruct` structure for the Identify cluster
  - Memory for tables or any other resources, if required by the cluster creation function

Note: If a custom endpoint is to co-exist with a device endpoint, the endpoints can share the structures for the clusters that they have in common. Therefore, it is not necessary to define these cluster structures for the custom endpoint, since they already exist for the device endpoint.

Revision History

Version	Date	Comments
1.0	10-June-2013	First release
1.1	22-Oct-2013	Poll Control and Power Profile clusters added

Important Notice

NXP Laboratories UK Ltd
(Formerly Jennic Ltd)
Furnival Street
Sheffield
S1 4QT
United Kingdom

Tel: +44 (0)114 281 2655
Fax: +44 (0)114 281 2951

For the contact details of your local NXP office or distributor, refer to:

www.nxp.com

For online support resources, visit the Wireless Connectivity TechZone:

www.nxp.com/techzones/wireless-connectivity