Zigbee stack API overview

This page provides an overview of the stack ZCL, commissioning, and security APIs.

Note

As of version 3.11.1.5, the ZBOSS stack supports Zigbee Cluster Library ver. 8. See ZCL8 support and compatibility modes for details.

Table of Contents

Common ZCL terms and definitions
Implementing a Zigbee end product with ZCL
Declaring attributes
Declaring cluster
Declaring custom cluster
Declaring cluster list
Declaring endpoint
Declaring simple descriptor
Declaring Zigbee device context
Declaring Zigbee device context with multiple endpoints
Registering device context
Configuring default ZCL command handler override
Executing interception mechanism
Implementing algorithm for overriding the handling of ZCL commands
Implementing Zigbee device callback
Sending ZCL commands
Parsing ZCL commands
ZCL8 support and compatibility modes
Modified clusters in ZCL8
ZCL command status changes
New ZBOSS APIs for ZCL8
Cluster revision
Local cluster revision
Peer cluster revision
Backward compatibility modes
Legacy mode
Automatic mode
Compatibility mode
Using backward compatibility modes
Support for Zigbee commissioning
Commissioning configuration sequence
Network configuration settings (general)
Network configuration settings (BDB-specific)
Stack commissioning start sequence
Zigbee commissioning event handling
BDB Commissioning API
Resetting to factory defaults
Resetting to factory defaults externally
Resetting to factory defaults with a local action
Security
Zigbee network security API

Common ZCL terms and definitions

The following table lists terms that are used in the **Zigbee Cluster Library specification <https://csa-iot.org/developer-resource/specifications-download-request/ >**.

Term	Definition
Attribute	A data entity that represents a physical quantity or state. This data is communicated to other devices through commands.
Cluster	A specification that defines one or more attributes, commands, behaviors, or dependencies. Cluster supports an independent utility or application function. The term can also be used for the implementation or the instance of the cluster specification on an endpoint.
Device	A specification that defines a unique device identifier and a set of mandatory and optional clusters to be implemented on a single endpoint. The term can also be used for the implementation or the instance of the device specification on an endpoint.
Endpoint	A device instance on a node.
Node	A testable implementation of a Zigbee application on a single stack with a single network address, and on a single network.

ZCL specification terms and definitions

The Zigbee application implements a node that handles endpoints. Every endpoint implements a set of clusters. Clusters implement the device functionality, such as attributes to store the state and the commands for various operations.

The ZCL specification is designed to minimize the consumption of flash and RAM by the user application. The application uses the API to declare a Zigbee device and to construct and parse ZCL commands.

Implementing a Zigbee end product with ZCL

Zigbee devices are characterized by a set of clusters that have mandatory and optional attributes. To save memory, make sure that only the required attributes and clusters are declared.

Application can either use predefined device declaration (for example, IAS Zone device) or define its own set of clusters and endpoints.

Zigbee API for code security installation

At the highest level, the Zigbee end product implementation consists of the following parts:

- Declaration of Zigbee device data structures:
  - Declaring attributes for each cluster (attribute lists),
  - Declaring cluster (or Declaring custom cluster),
  - Declaring cluster list for a device,
  - Declaring endpoint (including runtime context).
- Zigbee device executable code:
  - Registering device context (including optional implementation of a handler function for custom ZCL packets),
  - Configuring default ZCL command handler override,
  - Implementing Zigbee device callback (optional),
  - Sending ZCL commands and Parsing ZCL commands according to the application logic.

Declaring attributes

ZCL attributes are described according to the ZCL specification. The library declares attribute lists according to the mandatory attribute sets of the clusters. The ZCL attributes are defined in a form that allows modification.

Naming convention

Each attribute list declaration API has a name corresponding to ZB_ZCL_DECLARE_<CLUSTER_NAME>_CLUSTER_ATTRIB_LIST. The attribute list starts with ZB_ZCL_START_DECLARE_ATTRIB_LIST and ends with ZB_ZCL_FINISH_DECLARE_ATTRIB_LIST. You can add any number of additional attributes between these lines.

Example 1

The following macro defines the list of attributes for the Door Lock cluster:

```
#define ZB_ZCL_DECLARE_DOOR_LOCK_CLUSTER_ATTRIB_LIST(attr_list, \
  lock_state, \
 lock_type, \
 actuator_enabled) \
 ZB_ZCL_START_DECLARE_ATTRIB_LIST(attr_list) \
 ZB_ZCL_SET_ATTR_DESC(ZB_ZCL_ATTR_DOOR_LOCK_LOCK_STATE_ID, (lock_state)) \
 ZB_ZCL_SET_ATTR_DESC(ZB_ZCL_ATTR_DOOR_LOCK_LOCK_TYPE_ID, (lock_type)) \
 ZB_ZCL_SET_ATTR_DESC(ZB_ZCL_ATTR_DOOR_LOCK_ACTUATOR_ENABLED_ID, (actuator_enabled)) \
 ZB_ZCL_FINISH_DECLARE_ATTRIB_LIST
```

Example 2

The following code example defines global variables for the attributes and initiates them with the default values. It then declares the list of attributes for the On/Off Switch configuration cluster.

```
zb_uint8_t attr_switch_type = ZB_ZCL_ON_OFF_SWITCH_CONFIGURATION_SWITCH_TYPE_TOGGLE;
zb_uint8_t attr_switch_actions = ZB_ZCL_ON_OFF_SWITCH_CONFIGURATION_SWITCH_ACTIONS_DEFAULT_VALUE;

ZB_ZCL_DECLARE_ON_OFF_SWITCH_CONFIGURATION_ATTRIB_LIST(switch_cfg_attr_list,
 &attr_switch_type,
 &attr_switch_actions);
```

The `ZB_ZCL_DECLARE_ON_OFF_SWITCH_CONFIGURATION_ATTRIB_LIST` macro is defined in the `zb_zcl_on_off_switch.h` file. This cluster supports two attributes: `switch_type` and `switch_actions`.

Declaring cluster

The SDK provides more than 30 implemented Zigbee clusters that are ready for the user application.

To use a cluster, the application must declare it as part of the Zigbee device that is being implemented. Each cluster description is associated with a Zigbee device and includes the following data:

- Cluster ID (according to the Zigbee Cluster Library specification);
- Attribute list (according to Declaring attributes – the cluster declaration keeps both a pointer to the list of attributes and an attribute count);
- Cluster role (according to the Zigbee Cluster Library specification);

- Manufacturer specific code.

The cluster description data is stored using zb_zcl_cluster_desc_t type. The SDK provides several APIs to fill in cluster descriptors while creating the cluster list declaration.

Declaring custom cluster

When a non-standard cluster is required, implement it in the following manner:

1. Create the cluster headers with the name corresponding to the following template: `zb_zcl_<cluster_name>.h`
2. In the cluster headers, declare the required attributes with read, write, or reporting functionality.

> **Note**
> Refer to `zb_zcl_pres_measurement.h` for an example of the configured cluster header. -# Create the cluster implementation with the name corresponding to the following template: `zb_zcl_<cluster_name>.c`

3. In the created cluster implementation .c file, implement parsers for the required commands.

> **Note**
> Refer to `zb_zcl_pres_measurement.c` for an example of the cluster commands parser implementation. -# Add the new cluster to the device declaration header.

4. If you need specific commands in the new cluster, register proper functions using zb_zcl_add_cluster_handlers.

> **Note**
> See `zb_zcl_pres_measurement.c` for an example. -# In the created cluster implementation .c file, handle these custom commands. The commands are processed when the application layer returns ZB_TRUE.

Declaring cluster list

All the available Zigbee device functions are defined by a set of supported clusters. The cluster list is an array of the zb_zcl_cluster_desc_t type. For standard Zigbee devices, the source code contains a set of declaration APIs for cluster lists. A cluster list can be modified by adding or removing clusters.

> **Note**
> All cluster attributes in the cluster list must be declared. Missing or placing NULL instead of any attribute from the list causes errors during compilation or leads to inconsistent application behavior.

Naming convention
Each cluster list declaration API has a name corresponding to ZB_HA_DECLARE_<DEVICE_NAME>_CLUSTER_LIST.

Example
List declaration for a dimmable light (a standard Zigbee device):

```
ZB_HA_DECLARE_DIMMABLE_LIGHT_CLUSTER_LIST(cluster_list_name, basic_attr_list, identify_attr_list,
        groups_attr_list, scenes_attr_list, on_off_attr_list, level_control_attr_list)
```

> **Note**
> ZB_HA_DECLARE_DIMMABLE_LIGHT_CLUSTER_LIST() is defined in `zb_ha_dimmable_light.h`. Whenever the dimmable light utilizes Basic, Identify, Groups, Scenes, On/Off, and Level Control clusters in a server role, the device takes attribute lists for all of them.

Declaring endpoint

An endpoint or a set of endpoints fully describes a Zigbee device. The endpoint declaration in the user application finalizes the logical description of the Zigbee device. To implement ZCL features, each Zigbee device must have at least one endpoint declared besides the built-in one (with `ep_id` parameter value set to 0). This declared endpoint will be stored in the zb_af_endpoint_desc_t type endpoint list.

Each endpoint description includes the following data:

- Endpoint ID,

- Profile ID,
- Cluster list,
- Simple descriptor,
- Internal runtime data.

The endpoint description data is stored using the type zb_af_endpoint_desc_t. The SDK provides several APIs to fill in endpoint descriptors while performing a Zigbee device declaration for each supported Zigbee device. Additionally, the API declares a simple descriptor. The endpoint list declaration API is provided in the source code and can be modified depending on the application.

Naming convention
Each endpoint declaration API has a name corresponding to ZB_HA_DECLARE_<DEVICE_NAME>_EP.

Example
Endpoint declaration for a dimmable light (a standard Zigbee device):

```
ZB_HA_DECLARE_DIMMABLE_LIGHT_EP(ep_name, ep_id, cluster_list)
```

Declaring simple descriptor

The simple descriptor contains specific information for each endpoint contained in a node. It is mandatory for each endpoint present in the node.

The API declares the simple descriptor for each standard Zigbee device and embeds the related calls into the endpoint declaration APIs. However, if modifications to the standard device are requested, the simple descriptor declaration is provided in the source code and you can modify it according to the application needs.

Technically speaking, the simple descriptor is a variable of the type ZB_AF_SIMPLE_DESC_TYPE(in_clust_count, out_clust_count). A pointer to this variable is stored in the endpoint descriptor. This macro is used to create a full type name at compilation time. This is because the simple descriptor contains a list of in and out (client/server) clusters, and the list size varies for different devices. To minimize the RAM usage, the specified values of in and out clusters are used to declare an array of a size required to store all cluster IDs.

There is a base type declared for the simple descriptor: `ZB_AF_SIMPLE_DESC_TYPE(1, 1)`. This type is used to cast all the user-specific simple descriptor types at runtime.

Naming convention
Each simple descriptor declaration API has a name corresponding to ZB_HA_DECLARE_<DEVICE_NAME>_SIMPLE_DESC.

APIs
The APIs for managing simple descriptor types are the following:

- New type declaration: ZB_DECLARE_SIMPLE_DESC(in_clust_count, out_clust_count)
- Reference to the type name: ZB_AF_SIMPLE_DESC_TYPE(in_clust_count, out_clust_count)

Declaring Zigbee device context

The Zigbee device context aggregates the list of endpoints and runtime data, for example storage for reporting configuration information and Level Control context information. The application is responsible for providing enough storage for reporting and Level Control context data for all the endpoints that it declares. For standard Zigbee devices, it is the API that declares device context together with storage for all the context data.

Naming convention
Each device declaration API has a name corresponding to ZB_HA_DECLARE_<DEVICE_NAME>_CTX.

APIs
To modify the existing device context declaration or define a new one, use the API call ZBOSS_DECLARE_DEVICE_CTX_1_EP(device_ctx_name, ep_name).

Declaring Zigbee device context with multiple endpoints

The stack provides the ability to declare one or multiple endpoints for a device.

To declare a multiple endpoint device, the application must call `ZBOSS_DECLARE_DEVICE_CTX_<N>_EP`, where N is the number of endpoints.

The stack provides the default macros for declaring not more than 4 endpoints for a device. If the application requires a higher number of endpoints, use one of the following elements for declaring the desired device context:

- ZBOSS_DECLARE_DEVICE_CTX_3_EP(device_ctx_name, ep1_name, ep2_name, ep3_name),
- ZBOSS_DECLARE_DEVICE_CTX_4_EP(device_ctx_name, ep1_name, ep2_name, ep3_name, ep4_name)

For instance, ZBOSS_DECLARE_DEVICE_CTX_2_EP(device_ctx_name, ep1_name, ep2_name) declares the device context with two different endpoint descriptors (ep1_name, ep2_name).

For more information on how to declare endpoint descriptors, refer to the section Declaring endpoint.

Registering device context

When the declaration procedures are finished, the device context must be registered with a call to ZB_AF_REGISTER_DEVICE_CTX. As part of this call, two callbacks can be registered in the application:

- ZCL packet handler – registered with a call to ZB_AF_SET_ENDPOINT_HANDLER().
  - The handler allows for application-specific handling of the incoming ZCL messages. For more details, see Configuring default ZCL command handler override.
- Zigbee device callback – registered with a call to ZB_ZCL_REGISTER_DEVICE_CB().
  - The Zigbee device callback is a multipurpose callback used as a common entry point to a user-specific handler implementation for different cases. It covers attribute changes, device identification, actions specific to poll control, OTA firmware upgrade, and others. For more details, see Implementing Zigbee device callback.

Configuring default ZCL command handler override

The stack implements the default handling of all the ZCL commands for all supported clusters, but is also provides a mechanism to override the default settings by intercepting the default ZCL command processing and implementing a custom, application-specific command handling.

Executing interception mechanism

To execute this mechanism, the ZCL packet handler callback must be first implemented and registered by the application (see Registering device context). After the ZCL packet handler is registered, it is called for each new incoming ZCL command. This happens before the default ZCL handler is called.

- If the application-specific ZCL packet handler returns ZB_FALSE, the packet is not processed by the application and the default ZCL handling is applied.
- If the application does handle the packet and no default processing is needed, the packet handler returns ZB_TRUE. This means that the packet is processed.

Note

- If the application started parsing packets and modified a packet (header or tail bytes have been cut or byte order has been inverted), it must finalize the packet processing. Otherwise, the default handler can fail.
- If the application handles the received packet, it is responsible for sending a response (if needed) and managing the memory buffer that stores the original ZCL packet.

The most common commands processed by the application are responses to the requests sent, including Read attribute response, Write attribute response, and Configure reporting response.

APIs

The SDK comes with API macros for handling the received packets, for example ZB_ZCL_GENERAL_GET_NEXT_READ_ATTR_RES(), ZB_ZCL_GET_NEXT_WRITE_ATTR_RES(), and ZB_ZCL_GENERAL_GET_NEXT_CONFIGURE_REPORTING_RES(). Each packet-parsing API call can modify the ZCL packet it is parsing "in place". For more details on the parsing API, see ZCL commands shared by all clusters in the ZCL API documentation.

Macro	Parameters	Return value	Description
ZB_AF_SET_ENDPOINT_HANDLER(endpoint, handler)	`endpoint` – Endpoint number. `handler` – Pointer to a function of the type zb_device_handler_t.	None.	Register ZCL packet handler for the specific endpoint.
zb_device_handler_t(zb_uint8_t param)	zb_uint8_t `param` – ID of the buffer with the incoming ZCL command.	ZB_TRUE – If the ZCL command was processed by the application. ZB_FALSE – The ZCL command will be proceeded by the stack.	ZCL packet processing callback.

Processing API calls (selection)

Note
Registering device context is required before calling ZB_AF_SET_ENDPOINT_HANDLER().

Implementing algorithm for overriding the handling of ZCL commands

To process an incoming ZCL command, implement the following algorithm in the ZCL packet handler callback:

1. Extract the incoming ZCL command information from the incoming buffer:
```
zb_bufid_t zcl_cmd_buf = param;
zb_zcl_parsed_hdr_t *cmd_info = ZB_BUF_GET_PARAM(zcl_cmd_buf, zb_zcl_parsed_hdr_t);
```
2. Check command direction (the `cmd_direction` field of the `zb_zcl_parsed_hdr_t` structure):
    - towards a server: ZB_ZCL_FRAME_DIRECTION_TO_SRV, or
    - towards a client: ZB_ZCL_FRAME_DIRECTION_TO_CLI.
3. Check whether the command is general or cluster-specific.
    - When the command is general, the `is_common_command` field of the zb_zcl_parsed_hdr_t structure is set to 1.
    - When the command is cluster-specific, the `is_common_command` field is set to 0.
4. Check the designated cluster ID: `cmd_info -> cluster_id` field of the zb_zcl_parsed_hdr_t structure.
5. Check the command ID: `cmd_info -> cmd_id` field of the zb_zcl_parsed_hdr_t structure.
    - The general ZCL command identifiers are listed in the zb_zcl_cmd_t enumeration.
    - Cluster-specific command identifiers are defined in the corresponding cluster headers.
6. Depending on the check results:
    - Handle the command.
    - Let the stack do the handling.

If the incoming command was processed by the application:

- the buffer must be released by zb_buf_free(zb_bufid_t bufid) (if it was not reused),
- the callback must return ZB_TRUE.

Otherwise, return ZB_FALSE to indicate to the stack to perform the default command processing.

Implementing Zigbee device callback

Zigbee device callback is an optional callback, but in case of usage ZCL clusters from the table below, implementing of the Zigbee device callback becomes mandatory. It is called by the stack's default ZCL cluster handlers to notify the application about a certain event. The event can be caused either by the incoming ZCL command or by the internal stack logic.

Note

> When using ZCL clusters from the cluster callback table, implementing the Zigbee device callback becomes mandatory.

The "device" word in the callback name indicates that the application must react as a real device. For example, change the brightness level of the lamp on ZB_ZCL_LEVEL_CONTROL_SET_VALUE_CB_ID callback.

With a call to ZB_ZCL_REGISTER_DEVICE_CB(), the application can register a callback function that takes the reference to a buffer as the input parameter. This buffer carries the parameter zb_zcl_device_callback_param_t (see Zigbee stack memory management subsystem for information about how to get a parameter from the buffer).

To check what action triggered the callback, refer to the field `device_cb_id` of the received parameter. All the cases are listed in the zb_zcl_device_callback_id_t enumeration. Do not perform any blocking operations during the Zigbee device callback implementation. The memory buffer passed as a parameter to this callback must neither be released nor reused by the application because it is managed by the ZCL subsystem.

Examples

A few examples of device callbacks called by the ZBOSS stack:

- Attribute value change ZB_ZCL_SET_ATTR_VALUE_CB_ID :
  - by Write Attribute,
  - by special ZCL command,
  - by scene change.
- Level value change: ZB_ZCL_LEVEL_CONTROL_SET_VALUE_CB_ID.
- Intermediate event in OTA processing: ZB_ZCL_OTA_UPGRADE_VALUE_CB_ID.

Cluster callback table

Cluster	Callbacks list
ZCL On/Off cluster (Server)	ZB_ZCL_ON_OFF_WITH_EFFECT_VALUE_CB_ID
ZCL Shade Configuration cluster (Server)	ZB_ZCL_SHADE_SET_VALUE_CB_ID
^	ZB_ZCL_SHADE_GET_VALUE_CB_ID
ZCL Identify cluster (Server)	ZB_ZCL_IDENTIFY_EFFECT_CB_ID
ZCL Level control cluster (Server)	ZB_ZCL_LEVEL_CONTROL_SET_VALUE_CB_ID
ZCL Basic cluster (Server)	ZB_ZCL_BASIC_RESET_CB_ID
ZCL Thermostat cluster (Server)	ZB_ZCL_THERMOSTAT_VALUE_CB_ID
ZCL Poll Control cluster (Server)	ZB_ZCL_POLL_CONTROL_CHECK_IN_CLI_CB_ID
ZCL IAS Zone cluster (Server)	ZB_ZCL_IAS_ZONE_ENROLL_RESPONSE_VALUE_CB_ID
ZCL IAS WD cluster (Server)	ZB_ZCL_IAS_WD_START_WARNING_VALUE_CB_ID
^	ZB_ZCL_IAS_WD_SQUAWK_VALUE_CB_ID
ZCL IAS ACE cluster (Server)	ZB_ZCL_IAS_ACE_ARM_CB_ID
^	ZB_ZCL_IAS_ACE_BYPASS_CB_ID
^	ZB_ZCL_IAS_ACE_EMERGENCY_CB_ID
^	ZB_ZCL_IAS_ACE_FIRE_CB_ID
^	ZB_ZCL_IAS_ACE_PANIC_CB_ID
^	ZB_ZCL_IAS_ACE_GET_PANEL_STATUS_CB_ID
^	ZB_ZCL_IAS_ACE_GET_BYPASSED_ZONE_LIST_CB_ID
^	ZB_ZCL_IAS_ACE_GET_ZONE_STATUS_CB_ID
ZCL IAS ACE cluster (Client)	ZB_ZCL_IAS_ACE_ARM_RESP_CB_ID
^	ZB_ZCL_IAS_ACE_GET_ZONE_ID_MAP_RESP_CB_ID
^	ZB_ZCL_IAS_ACE_GET_ZONE_INFO_RESP_CB_ID
^	ZB_ZCL_IAS_ACE_ZONE_STATUS_CHANGED_CB_ID
^	ZB_ZCL_IAS_ACE_PANEL_STATUS_CHANGED_CB_ID
^	ZB_ZCL_IAS_ACE_GET_PANEL_STATUS_RESP_CB_ID
^	ZB_ZCL_IAS_ACE_SET_BYPASSED_ZONE_LIST_CB_ID
^	ZB_ZCL_IAS_ACE_BYPASS_RESP_CB_ID
^	ZB_ZCL_IAS_ACE_GET_ZONE_STATUS_RESP_CB_ID
ZCL OTA Upgrade cluster (Client)	ZB_ZCL_OTA_UPGRADE_VALUE_CB_ID
ZCL OTA Upgrade cluster (Server)	ZB_ZCL_OTA_UPGRADE_SRV_QUERY_IMAGE_CB_ID
^	ZB_ZCL_OTA_UPGRADE_SRV_UPGRADE_STARTED_CB_ID
^	ZB_ZCL_OTA_UPGRADE_SRV_UPGRADE_ABORTED_CB_ID
^	ZB_ZCL_OTA_UPGRADE_SRV_UPGRADE_END_CB_ID
EN50523 Appliance events and alerts cluster (Server)	ZB_ZCL_EN50523_APPL_EV_AND_ALERTS_GET_ALERTS_CB_ID
EN50523 Appliance events and alerts cluster (Client)	ZB_ZCL_EN50523_APPL_EV_AND_ALERTS_GET_ALERTS_RESP_CB_ID
^	ZB_ZCL_EN50523_APPL_EV_AND_ALERTS_ALERTS_NOTIFICATION_CB_ID
^	ZB_ZCL_EN50523_APPL_EV_AND_ALERTS_EVENT_NOTIFICATION_CB_ID
ZCL Price cluster (Server)	ZB_ZCL_PRICE_GET_CURRENT_PRICE_CB_ID
^	ZB_ZCL_PRICE_GET_SCHEDULED_PRICES_CB_ID
^	ZB_ZCL_PRICE_GET_TIER_LABELS_CB_ID
^	ZB_ZCL_PRICE_PRICE_ACK_CB_ID
ZCL Price cluster (Client)	ZB_ZCL_PRICE_PUBLISH_PRICE_CB_ID

^	ZB_ZCL_PRICE_PUBLISH_TIER_LABELS_CB_ID
ZCL Demand Response and Load Control (DRLC) cluster (Server)	ZB_ZCL_DRLC_REPORT_EVENT_STATUS_CB_ID
^	ZB_ZCL_DRLC_GET_SCHEDULED_EVENTS_CB_ID
ZCL Demand Response and Load Control (DRLC) cluster (Client)	ZB_ZCL_DRLC_LOAD_CONTROL_EVENT_CB_ID
^	ZB_ZCL_DRLC_CANCEL_LOAD_CONTROL_EVENT_CB_ID
^	ZB_ZCL_DRLC_CANCEL_ALL_LOAD_CONTROL_EVENTS_CB_ID
ZCL Messaging cluster (Server)	ZB_ZCL_MESSAGING_MSG_CONFIRMATION_CB_ID
^	ZB_ZCL_MESSAGING_GET_LAST_MSG_CB_ID
ZCL Messaging cluster (Client)	ZB_ZCL_MESSAGING_CANCEL_MSG_CB_ID
^	ZB_ZCL_MESSAGING_DISPLAY_MSG_CB_ID
ZCL Tunneling cluster (Server)	ZB_ZCL_TUNNELING_REQUEST_TUNNEL_CB_ID
^	ZB_ZCL_TUNNELING_TRANSFER_DATA_CLI_CB_ID
^	ZB_ZCL_TUNNELING_TRANSFER_DATA_ERROR_CLI_CB_ID
^	ZB_ZCL_TUNNELING_CLOSE_TUNNEL_CB_ID
ZCL Tunneling cluster (Client)	ZB_ZCL_TUNNELING_REQUEST_TUNNEL_RESPONSE_CB_ID
^	ZB_ZCL_TUNNELING_TRANSFER_DATA_SRV_CB_ID
^	ZB_ZCL_TUNNELING_TRANSFER_DATA_ERROR_SRV_CB_ID
ZCL Metering Cluster (Server)	ZB_ZCL_METERING_GET_PROFILE_CB_ID
^	ZB_ZCL_METERING_REQUEST_FAST_POLL_MODE_CB_ID
^	ZB_ZCL_METERING_GET_SNAPSHOT_CB_ID
^	ZB_ZCL_METERING_GET_SAMPLED_DATA_CB_ID
ZCL Metering Cluster (Client)	ZB_ZCL_METERING_GET_PROFILE_RESPONSE_CB_ID
^	ZB_ZCL_METERING_REQUEST_FAST_POLL_MODE_RESPONSE_CB_ID
^	ZB_ZCL_METERING_PUBLISH_SNAPSHOT_CB_ID
^	ZB_ZCL_METERING_GET_SAMPLED_DATA_RESPONSE_CB_ID
ZCL Window Covering cluster (Server)	ZB_ZCL_WINDOW_COVERING_UP_OPEN_CB_ID
^	ZB_ZCL_WINDOW_COVERING_DOWN_CLOSE_CB_ID
^	ZB_ZCL_WINDOW_COVERING_STOP_CB_ID
^	ZB_ZCL_WINDOW_COVERING_GO_TO_LIFT_PERCENTAGE_CB_ID
^	ZB_ZCL_WINDOW_COVERING_GO_TO_TILT_PERCENTAGE_CB_ID
ZCL DoorLock cluster (Server)	ZB_ZCL_DOOR_LOCK_LOCK_DOOR_CB_ID
^	ZB_ZCL_DOOR_LOCK_UNLOCK_DOOR_CB_ID
ZCL DoorLock cluster (Client)	ZB_ZCL_DOOR_LOCK_LOCK_DOOR_RESP_CB_ID
^	ZB_ZCL_DOOR_LOCK_UNLOCK_DOOR_RESP_CB_ID
ZCL Alarms cluster (Server)	ZB_ZCL_ALARMS_RESET_ALARM_CB_ID
^	ZB_ZCL_ALARMS_RESET_ALL_ALARMS_CB_ID
ZCL Alarms cluster (Client)	ZB_ZCL_ALARMS_ALARM_CB_ID
ZCL Work With All Hubs (WWAH) cluster (Server)	ZB_ZCL_WWAH_ENABLE_APP_EVENT_RETRY_ALGORITHM_CB_ID
^	ZB_ZCL_WWAH_DISABLE_APP_EVENT_RETRY_ALGORITHM_CB_ID
^	ZB_ZCL_WWAH_DEBUG_REPORT_QUERY_CB_ID
^	ZB_ZCL_WWAH_SET_IAS_ZONE_ENROLLMENT_METHOD_CB_ID
ZCL Control4 Network cluster (Server)	ZB_ZCL_CONTROL4_NETWORK_ZAP_INFO_CB_ID
ZCL Scenes cluster (Server)	ZB_ZCL_SCENES_ADD_SCENE_CB_ID
^	ZB_ZCL_SCENES_STORE_SCENE_CB_ID

^	ZB_ZCL_SCENES_VIEW_SCENE_CB_ID
^	ZB_ZCL_SCENES_REMOVE_SCENE_CB_ID
^	ZB_ZCL_SCENES_REMOVE_ALL_SCENES_CB_ID
^	ZB_ZCL_SCENES_RECALL_SCENE_CB_ID
^	ZB_ZCL_SCENES_GET_SCENE_MEMBERSHIP_CB_ID
^	ZB_ZCL_SCENES_INTERNAL_REMOVE_ALL_SCENES_ALL_ENDPOINTS_CB_ID
^	ZB_ZCL_SCENES_INTERNAL_REMOVE_ALL_SCENES_ALL_ENDPOINTS_ALL_GROUPS_CB_ID

ZCL cluster callbacks

Sending ZCL commands

The SDK provides API calls for sending and parsing ZCL packets for general command frames and cluster-specific commands.

Naming convention

The naming convention is not strict. Usually, it is composed of the following elements:

- cluster name (for general commands, it is replaced with GENERAL),
- verb describing an action,
- command name.

Macro example	Cluster name	Action verb	Command
ZB_ZCL_ON_OFF_SEND_TOGGLE_REQ()	On/Off	Send	Toggle
ZB_ZCL_IAS_ZONE_SEND_STATUS_CHANGE_NOTIFICATION_REQ()	IAS zone	Send	Zone notification

ZB_ZCL_GENERAL_INIT_READ_ATTR_REQ()
ZB_ZCL_GENERAL_ADD_ID_READ_ATTR_REQ()
ZB_ZCL_GENERAL_SEND_READ_ATTR_REQ()
	(general commands)	Initialize
		Add
		Send
			Read Attribute

Sending ZCL commands: Naming convention examples

API command structure

The command can have a predefined or a variable size:

- If a command has a predefined size, it is composed and sent with one API call.
  - The Toggle command is an example of this type of command.
- If a command has a variable size, it is composed dynamically at runtime.
  - The Read Attribute command is an example of this type of command. An application can call ZB_ZCL_GENERAL_ADD_ID_READ_ATTR_REQ() several times to add more attribute IDs to read.

To construct a manufacturer-specific command (or a standard command that is not supported by the API yet), use the low-level API:

- A function to initialize a ZCL packet: zb_zcl_start_command_header()
- Primitives to fill in the packet data:
  - ZB_ZCL_PACKET_PUT_DATA8
  - ZB_ZCL_PACKET_PUT_DATA16
  - ZB_ZCL_PACKET_PUT_DATA32
  - ZB_ZCL_PACKET_PUT_DATA_IEEE
  - ZB_ZCL_PACKET_PUT_DATA_N
- A function to finalize the packet and to send it: zb_zcl_finish_and_send_packet()

Optionally, a callback can be initiated by including a callback function into a packet. On return, it contains the following parameters:

- at the begining of the buffer (zb_buf_begin(zb_bufid_t bufid)): the original ZCL request

- at the end of the buffer (ZB_BUF_GET_PARAM(buf, type)): status of the transmission (zb_zcl_command_send_status_t)

Parsing ZCL commands

The API for parsing the received ZCL packets for general command frames and cluster-specific commands is similar to the API for Sending ZCL commands.

- In most cases, the stack automatically parses the incoming command and calls the corresponding ZCL callback (if needed).
  - This is true for all mandatory Request commands (both general and cluster-specific, for example Read Attributes Request, On/Off Cluster's On command, etc.).
- For most of the attribute-changing commands, the stack uses one unified SET_ATTRIBUTE callback with specified cluster ID, attribute ID and a new value.
  - This is true, for example, for On/Off, Level Control, Color Control etc.
  - Otherwise, the application needs to parse this command in the zcl_specific_cmd_handler.
    - This is true for most of the responses, when the request is generated from the application and is not related to the stack logic (for example, Read Attributes Response, IAS Zone Status Change Notification, etc.).

Note
All OTA commands are parsed by the stack. In general, if a command has a corresponding ZCL callback, it is parsed by the stack.

Naming convention
The API command name is composed of the following elements:

- cluster name (for general commands, it is replaced with GENERAL),
- verb describing an action,
- command name.

Macro example	Cluster name	Action verb	Command	Additional information
ZB_ZCL_IAS_ZONE_GET_STATUS_CHANGE_NOTIFICATION_REQ()	IAS zone	Get	Zone notification	Fills in a variable of the zb_zcl_ias_zone_status_change_not_t type.
ZB_ZCL_GENERAL_GET_NEXT_READ_ATTR_RES()	(general command)	Get	Read Attribute	Parses the Read Attribute response.

Parsing ZCL commands: Naming convention examples

The parsing API is not provided for commands without a payload, for example the Toggle command in the On/Off cluster. If the received packet is parsed manually, use ZB_ZCL_FIX_ENDIAN API to keep the right endianness in the packet.

ZCL8 support and compatibility modes

The following changes were introduced in Zigbee Cluster Library ver. 8 (ZCL8) compared with the Zigbee Cluster Library ver. 7 (ZCL7):

- Modified clusters with additional fields and status codes.
- Modified ZCL error statuses of several ZCL commands.
- Updated cluster revision numbers.

These changes were introduced to be compliant with the new version of the ZCL specification.

Additionally, the ZBOSS stack introduced different Backward compatibility modes to ensure the interoperability with devices based on the previous revisions of ZCL.

Modified clusters in ZCL8

The following table lists ZCL cluster names alongside their ZBOSS cluster identifiers.

Cluster name	ZBOSS ZCL identifier	Changes
Color control cluster	ZB_ZCL_CLUSTER_ID_COLOR_CONTROL	Updated range of values valid for color temperature attributes. Also, `OptionsOverride` and `OptionsMask` are now mandatory.
Level control cluster	ZB_ZCL_CLUSTER_ID_LEVEL_CONTROL	`OptionsOverride` and `OptionsMask` are now mandatory.
Scenes cluster	ZB_ZCL_CLUSTER_ID_SCENES	Status codes updated, including status code update for out of range Group ID field value.
Demand-Response cluster	ZB_ZCL_CLUSTER_ID_DRLC	Status codes updated and added fields.
Alarms cluster	ZB_ZCL_CLUSTER_ID_ALARMS	Status codes updated.
Appliance events and alerts cluster	ZB_ZCL_CLUSTER_ID_APPLIANCE_EVENTS_AND_ALERTS	Status codes updated.
Over The Air cluster	ZB_ZCL_CLUSTER_ID_OTA_UPGRADE	Status codes updated. Also, updated the valid range of the Image type values.
Door lock cluster	ZB_ZCL_CLUSTER_ID_DOOR_LOCK	Status codes updated.
Groups cluster	ZB_ZCL_CLUSTER_ID_GROUPS	Status codes updated, including status code update for out of range Group ID field value.
IAS ACE cluster	ZB_ZCL_CLUSTER_ID_IAS_ACE	Status codes updated.
IAS WD cluster	ZB_ZCL_CLUSTER_ID_IAS_WD	Status codes updated.
IAS Zone cluster	ZB_ZCL_CLUSTER_ID_IAS_ZONE	Status codes updated.
Identify cluster	ZB_ZCL_CLUSTER_ID_IDENTIFY	Status codes updated.
Window covering cluster	ZB_ZCL_CLUSTER_ID_WINDOW_COVERING	Status codes updated.
On/Off cluster	ZB_ZCL_CLUSTER_ID_ON_OFF	Status codes updated.
Poll control cluster	ZB_ZCL_CLUSTER_ID_POLL_CONTROL	Status codes updated.
Works with All Hubs cluster	ZB_ZCL_CLUSTER_ID_WWAH	Status codes updated.
Thermostat cluster	ZB_ZCL_CLUSTER_ID_THERMOSTAT	Status codes updated.
Time cluster	ZB_ZCL_CLUSTER_ID_TIME	Time attribute can now be read-only depending on the `TimeStatus` attribute value.

ZCL command status changes

The following table lists command status changes introduced in the ZCL8 specification.

Pre-ZCL8 status codes	Status code ID	Changes in ZCL8
UNSUP_CLUSTER_COMMAND	0x81	Renamed to UNSUP_COMMAND.
UNSUP_GENERAL_COMMAND	0x82	Use UNSUP_COMMAND.
UNSUP_MANUF_CLUSTER_COMMAND	0x83	Use UNSUP_COMMAND.
UNSUP_MANUF_GENERAL_COMMAND	0x84	Use UNSUP_COMMAND.
DUPLICATE_EXISTS	0x8a	Use SUCCESS (0x00).
WRITE_ONLY	0x8f	Use NOT_AUTHORIZED(0x7e).
INCONSISTENT_STARTUP_STATE	0x90	Use FAILURE (0x01).
DEFINED_OUT_OF_BAND	0x91	Use FAILURE (0x01).
INCONSISTENT	0x92	RESERVED in ZCL8. Do not use.
ACTION_DENIED	0x93	Use FAILURE (0x01).
HARDWARE_FAILURE	0xc0	Use FAILURE (0x01).
SOFTWARE_FAILURE	0xc1	Use FAILURE (0x01).
CALIBRATION_ERROR	0xc2	RESERVED in ZCL8. Do not use.
LIMIT_REACHED	0xc4	Use SUCCESS (0x00). In ZCL6, this didn't exist.

If you want ZBOSS to translate these status codes, you need to enable the backward compatibility status code mode and use the zb_zcl_set_backward_compatible_statuses_mode() API. For more information, see the Backward compatibility modes section.

New ZBOSS APIs for ZCL8

The ZBOSS ZCL8 implementation extends the ZCL API with new requests with the `_ZCL8` suffix. For example:

- ZB_ZCL_LEVEL_CONTROL_SEND_MOVE_WITH_ON_OFF_REQ() is a pre-ZCL8 API.
- ZB_ZCL_LEVEL_CONTROL_SEND_MOVE_WITH_ON_OFF_REQ_ZCL8() is the ZCL8 API.

These requests conform to the ZCL8 specification and may require passing additional fields, which are needed to define ZCL8-compatible commands.

Although not all ZCL commands require updates (see Modified clusters in ZCL8 for the list of modified clusters), the API calls with the `_ZCL8` suffix are provided for all requests. To use them, make sure the header `zboss_api_zcl8.h` is included in your application.

Cluster revision

The `ClusterRevision` is a global attribute, mandatory for all Zigbee clusters since ZCL6. It is described in the section 2.3.4.5 of the ZCL specification (Global Attributes) and in the Zigbee Application Architecture document (ID 13-0589-13).

The cluster revision attribute value can range from 1 to 4, depending on the number of updates the cluster has received. For example, if a ZCL6 cluster was updated in both ZCL7 and ZCL8, its cluster revision will equal 3. Refer to the ZCL specification's revision history section for each cluster to check the latest revision number and the list of changes made in each cluster specification.

The currently supported revision of the cluster is specified in ZBOSS with a set of defines ending with `_CLUSTER_REVISION_DEFAULT`, one for each cluster.

The value of this attribute can be read locally or by remote devices (peers).

Local cluster revision

In the ZBOSS application, the value of the `ClusterRevision` attribute is assigned during the cluster declaration when `ZB_ZCL_DECLARE_<CLUSTER_NAME>_ATTRIB_LIST` is called. The default value for the cluster revision attribute is the highest revision supported by ZBOSS.

The `ClusterRevision` attribute value can be modified manually by using the ZB_ZCL_SET_ATTRIBUTE API. In this case, however, the revision value passed with the API should not be greater than the highest supported revision. You might want to have the application decrease the local cluster revision in order to work with legacy devices.

The local cluster revision can be obtained using the ZB_ZCL_GET_ATTRIBUTE_VAL_16 API.

If the application defines its own cluster declaration, make sure that it uses the updated API to define the cluster attribute list (ZB_ZCL_START_DECLARE_ATTRIB_LIST_CLUSTER_REVISION instead of the pre-ZCL8 ZB_ZCL_START_DECLARE_ATTRIB_LIST). Otherwise, the device will use the pre-ZCL8 implementation and revision and it will not be possible to update the cluster revision attribute value.

Peer cluster revision

The peer cluster revision is the revision of the cluster on a specific remote device. By using the peer revision, the application discovers which functionality is supported by the remote device.

The peer cluster revision can be obtained using the ZCL Read Attribute command. The following code sample initializes, composes, and sends a command for reading the revision of the given `cluster_id` of the device with the `dst_addr` address:

```
ZB_ZCL_GENERAL_INIT_READ_ATTR_REQ(buf, cmd_ptr, ZB_ZCL_ENABLE_DEFAULT_RESPONSE);
ZB_ZCL_GENERAL_ADD_ID_READ_ATTR_REQ(cmd_ptr, ZB_ZCL_ATTR_GLOBAL_CLUSTER_REVISION_ID);
ZB_ZCL_GENERAL_SEND_READ_ATTR_REQ(buf, cmd_ptr, dst_addr, dst_addr_mode, dst_ep, src_ep, profile_id,
       cluster_id, cb);
```

Backward compatibility modes

The ZBOSS stack implementation of ZCL8 comes with backward compatibility modes, which are meant to help with the migration process when upgrading applications to ZCL8. You can use these modes to avoid having to make changes to already released products.

By default, the ZBOSS implementation uses legacy status codes and the `ZB_ZCL_LEGACY_MODE` mode. This behavior can be changed manually. See Using backward compatibility modes for details.

Legacy mode

The legacy mode ensures the application compatibility with devices that are using an earlier version of ZCL. By default, the ZCL requests in this mode are sent in the pre-ZCL8 format.

If the application wants to send a command using the new ZCL8 format, it must meet the following conditions:

- The application must use the New ZBOSS APIs for ZCL8.
- The application must leave the cluster revision set to the default ZCL8-compatible value.

If the application uses the ZCL8 API and the cluster revision is set to a pre-ZCL8 value, the command format is downgraded automatically to match the revision format configured by the attribute. In most cases, the downgrade means cutting off all fields that are absent in the format specification for the given revision.

Here are some examples of how the legacy mode works:

- If the cluster revision attribute is set to any revision, and the application calls a *pre-ZCL8* API request, the request will be formatted according to the revision 1 format.
- If the cluster revision attribute value is 3, and the application calls ZBOSS *ZCL8* API request, the request will be formatted according to the cluster revision 3.
- If the cluster revision attribute is set to a *pre-ZCL8* value, for example to 1, and the application calls ZBOSS *ZCL8* API request, the request format will be downgraded to correspond to the revision 1 format as a result.

Automatic mode

The `ZB_ZCL_AUTO_MODE` mode is meant for automatic conversion of the application that uses ZBOSS pre-ZCL8 API to ZCL8. Any ZBOSS ZCL API will be either upgraded or downgraded to match the format given by the cluster revision attribute value.

- In case of an upgrade, the request fields will be set to their default values according to the ZCL specification.
- In case of a downgrade, all fields that are absent in the format specification for the given revision are cut.

Here are some examples of how the automatic mode works:

- If the cluster revision attribute value equals 3, the *pre-ZCL8* API request will be extended with new fields using their default values to match the revision 3 format.
- If the cluster revision attribute value equals 3 and the application calls a ZBOSS *ZCL8* API request, the request will be sent unchanged in the *ZCL8* format.
- If the cluster revision attribute is set to a *pre-ZCL8* value, for example 1, and the application calls a *pre-ZCL8* API request, the request will be sent unchanged in the *pre-ZCL8* format.
- If the cluster revision attribute is set to a *pre-ZCL8* value, for example 1, and the application calls a ZBOSS *ZCL8* API request, the request will be downgraded to the *pre-ZCL8* format.

Compatibility mode

Note

The ZB_ZCL_COMPATIBILITY_MODE mode must be supported in the application to match the ZCL specification.

The ZB_ZCL_COMPATIBILITY_MODE mode adds a mechanism for requesting revision of the peer (see Peer cluster revision). Based on this information, commands sent to the remote device by the application will be either upgraded or downgraded to match the format supported by the remote device.

To use this mode, you must extend your application to read and maintain cluster revision table and implement the full cluster revision API. The ZBOSS stack will call an application callback for each ZCL request, asking for the cluster revision that should be used for the command. The callback must be defined according to the following signature:

```
typedef zb_uint16_t (*zb_zcl_peer_revision_cb_t) (zb_ieee_addr_t ieee_addr, zb_uint16_t cluster_id,
       zb_uint8_t cluster_role, zb_uint8_t endpoint);
```

Here's an example implementation of zb_zcl_set_peer_revision_callback :

```
zb_uint16_t peer_revision_cb(zb_ieee_addr_t ieee_addr, zb_uint16_t cluster_id, zb_uint8_t cluster_role,
       zb_uint8_t endpoint)
{
 ...
 return peer_revision;
}

void zcl_attr_init(void)
{
 zb_ret_t ret;
 ...
 ret = zb_zcl_set_peer_revision_callback(peer_revision_cb);
 ...
}
```

The callback must return the ZB_ZCL_PEER_CLUSTER_REV_UNKNOWN value if the cluster revision is unknown. In such case, the minimal supported cluster revision will be used.

This mode is enabled automatically if the application sets the peer_revision_callback.

Using backward compatibility modes

Using ZCL backward compatibility modes requires both setting the status code mode and choosing the compatibility mode of ZCL requests. Both have to match the ZCL version used.

Setting the status code mode

To enable the status code mode, use the zb_zcl_set_backward_compatible_statuses_mode() API:

```
zb_ret_t ret = zb_zcl_set_backward_compatible_statuses_mode(ZB_ZCL_STATUSES_PRE_ZCL8_MODE);
```

The ZB_ZCL_STATUSES_PRE_ZCL8_MODE is always the default mode. It uses status codes from before ZCL command status changes. The information about the currently used status code mode can be obtained through a call to zb_zcl_get_backward_compatible_statuses_mode :

```
zb_uint8_t mode = zb_zcl_get_backward_compatible_statuses_mode();
```

You can change the default setting manually to use ZB_ZCL_STATUSES_ZCL8_MODE. In such case, the status codes are automatically translated based on the table in ZCL command status changes.

Setting the compatibility mode

To enable the ZBOSS ZCL8 backward compatibility modes, use the call to zb_zcl_set_backward_comp_mode and specify the compatibility mode you want to use:

```
zb_ret_t ret = zb_zcl_set_backward_comp_mode(ZB_ZCL_LEGACY_MODE);
```

ZB_ZCL_LEGACY_MODE is always the default mode. The information about the currently used mode can be obtained through a call to zb_zcl_get_backward_comp_mode :

```
zb_uint8_t mode = zb_zcl_get_backward_comp_mode();
```

The following table lists all of the compatibility modes and ZCL API usage with the corresponding resulting ZCL request format:

Backward compatibility mode	ZBOSS ZCL API version	Local cluster revision	Peer cluster revision	Resulting request format
`ZB_ZCL_LEGACY_MODE`	Pre-ZCL8	1	n/a	Legacy, unchanged due to ZCL API version used.
`ZB_ZCL_LEGACY_MODE`	Pre-ZCL8	default(3)	n/a	Legacy, unchanged due to ZCL API version used.
`ZB_ZCL_LEGACY_MODE`	ZCL8	1	n/a	Legacy, downgraded due to the local cluster revision.
`ZB_ZCL_LEGACY_MODE`	ZCL8	default(3)	n/a	ZCL8, unchanged due to ZCL API version used. ZCL8 allowed by the local cluster revision.
`ZB_ZCL_AUTO_MODE`	Pre-ZCL8	1	n/a	Legacy, upgrade to ZCL8 not allowed by the local cluster revision.
`ZB_ZCL_AUTO_MODE`	Pre-ZCL8	default(3)	n/a	ZCL8, upgrade allowed by the local cluster revision.
`ZB_ZCL_AUTO_MODE`	ZCL8	1	n/a	Legacy, downgraded to pre-ZCL8 due to the local cluster revision.
`ZB_ZCL_AUTO_MODE`	ZCL8	default(3)	n/a	ZCL8, unchanged due to ZCL API version used. ZCL8 allowed by the local cluster revision.
`ZB_ZCL_COMPATIBILITY_MODE`	Pre-ZCL8	1	1	Legacy, unchanged due to the local cluster revision.
`ZB_ZCL_COMPATIBILITY_MODE`	Pre-ZCL8	1	3	Legacy, unchanged due to the local cluster revision.
`ZB_ZCL_COMPATIBILITY_MODE`	Pre-ZCL8	default(3)	1	Legacy, unchanged due to the peer cluster revision.
`ZB_ZCL_COMPATIBILITY_MODE`	Pre-ZCL8	default(3)	3	ZCL8, upgrade allowed by both the local and the peer cluster revision.
`ZB_ZCL_COMPATIBILITY_MODE`	ZCL8	1	1	Legacy, downgraded to pre-ZCL8 due to both the local and the peer cluster revision.
`ZB_ZCL_COMPATIBILITY_MODE`	ZCL8	1	3	Legacy, downgraded to pre-ZCL8 due to the local cluster revision.
`ZB_ZCL_COMPATIBILITY_MODE`	ZCL8	default(3)	1	Legacy, downgraded to pre-ZCL8 due to the peer cluster revision.
`ZB_ZCL_COMPATIBILITY_MODE`	ZCL8	default(3)	3	ZCL8, unchanged due to both the local and the peer cluster revision.

Note
The cluster revision values in the table are provided as an example to demonstrate possible format conversions.

Support for Zigbee commissioning

The commissioning logic is based on Zigbee BDB specification, and is divided into the following types:

- Normal commissioning (network steering, formation, finding and binding),

Other types of commissioning like the legacy HA or the ZLL commissioning are supported, but not recommended.

You can configure commissioning by setting parameters before the start of the stack.

Commissioning configuration sequence

There are several commissioning parameters that can be assigned by the application or stored in NVRAM, or both. The default parameters are coded in the application firmware. Then they can be updated by the NVRAM content.

The commissioning configuration sequence is made of the following steps:

- Application calls ZB_INIT() to set some reasonable defaults of commissioning parameters.
- If needed, the application makes an API call to tune commissioning by changing the default values.
    - **Example 1**: set hard-coded long address (useful for debugging): `zb_set_long_address(g_ed_addr);`
    - **Example 2**: set a ZED at channel 2: `zb_set_network_ed_role(1l<<21);`
- The application calls zboss_start() or zboss_start_no_autostart().
    - The stack loads NVRAM and updates internal parameters, for example working channel, short address, binding information.
    - The stack loads the production configuration from the flash memory. This can possibly modify some settings related to the implementation of a Zigbee device.
    - The stack continues the starting process. If a device joins the network, the stack updates the short address and stores it in NVRAM.
- The application does some commission actions (like binding) at runtime, after successful start and join. The stack stores the binding information in NVRAM to be able to restore it at reboot.

The following example demonstrates the commissioning configuration sequence, with the long address updated by the long address defined in the production configuration block:

```
zb_ieee_addr_t g_ed_addr = {0x22, 0x22, 0x22, 0x22, 0x22, 0x22, 0x22, 0x22};

int main(void)
{
 ...
 ZB_INIT("on_off_switch_zed");

 zb_set_long_address(g_ed_addr);
 zb_set_network_ed_role(1l<<21);
 zb_set_rx_on_when_idle(ZB_FALSE);

...
 if (zboss_start() != RET_OK)
...
```

Network configuration settings (general)

The stack implements all Zigbee roles: a coordinator, a router, and an end device. Only one role can be defined in a Zigbee application. Switching Zigbee role at runtime is not supported.

Different libraries are provided for each role, and the application must be linked to them:

- For the coordinator or router: `libzboss.a`,
- For the end device: `libzboss.ed.a`.

See the following table for the overview of functions for setting the device role.

Macro	Parameters	Return value	Description
zb_set_network_coordinator_role(zb_uint32_t channel_mask)	`channel_mask` – mask used to define the primary channel set.	None.	Initiate device as a Zigbee coordinator.
zb_set_network_router_role(zb_uint32_t channel_mask)	`channel_mask` – mask used to define the primary channel set.	None.	Initiate device as a Zigbee router.
zb_set_network_ed_role(zb_uint32_t channel_mask)	`channel_mask` – mask used to define the primary channel set.	None.	Initiate device as a Zigbee end device.

Functions for setting the device role

For the complete list of settings, see the Application structure & commissioning start section in the Zigbee stack API reference documentation.

Network configuration settings (BDB-specific)

The `channel_mask` parameter is a bit field that defines the Zigbee channels to be used for the energy scan as the primary channel set. Bits from 11 to 26 are used to control the channel selection. Each bit enables or disables the corresponding channel. To set all channel masks (enable all channels), use the macro `ZB_TRANSCEIVER_ALL_CHANNELS_MASK`.

BDB has an abstraction of primary and secondary channel mask. By default, the secondary channel mask is the same, but you can define it separately by using the call zb_set_bdb_secondary_channel_set(zb_uint32_t channel_mask).

For details on the abstraction of primary and secondary channel mask and the complete network configuration API, see the official Zigbee BDB specification.

The following functions can modify the settings of the BDB channel:

- zb_set_bdb_primary_channel_set()
- zb_get_bdb_primary_channel_set()
- zb_set_bdb_secondary_channel_set()
- zb_get_bdb_secondary_channel_set()

To start the implicit BDB commissioning after the zboss_start() call, use the zb_set_bdb_commissioning_mode() function. Its parameter is a bitmask that specifies the BDB commissioning mode. See the official Zigbee BDB specification for detailed description. For more details about the API, see the BDB commissioning API section in the Zigbee stack API documentation.

Stack commissioning start sequence

You can use one of the following functions for the stack initialization:

- zboss_start() to initialize the radio and start commissioning according to parameters that you set after the initialization is complete.
- zboss_start_no_autostart() to delay commissioning until an action takes place.

The zboss_start_no_autostart() function does only minimal initialization of the stack framework, enough to have the multitasking and the memory management. The stack initializes its base structures, reads NVRAM, starts the scheduler, but does not initialize Zigbee data structures and radio. It then calls zboss_signal_handler() with ZB_ZDO_SIGNAL_SKIP_STARTUP signal. The application should later call ZBOSS commissioning initiation – bdb_start_top_level_commissioning(ZB_BDB_NETWORK_STEERING) that will trigger further commissioning. To avoid triggering commissioning at this point, call bdb_start_top_level_commissioning() with mask 0 that makes the stack proceed with the initialization without steering and commissioning. This option is recommended if:

- The application is doing some other hardware initialization before the start of commissioning and prefers to use the stack multitasking during this initialization (for example, read some sensor data through the serial port).
- The application does not start commissioning at power-up. Instead, it waits for a user action at every power-up (for example, a button press).

After starting zboss_main_loop or zboss_main_loop_iteration, the stack will interact with the user application using:

- stack signals during internal processes such as commissioning, leave, etc. – see zboss_signal_handler().
- ZCL callbacks (on incoming ZCL frames) – see Implementing Zigbee device callback.

Note

The stack provides you with the following ways for calling the main loop:

- zboss_main_loop() - by default.
- zboss_main_loop_iteration() - in case the user needs to include custom features.

Example 1

zboss_main_loop() implementation example:

```
if (zboss_start() == RET_OK)
 {
 zboss_main_loop();
 }
```

Example 2

zboss_main_loop_iteration() implementation example:

```
void my_main_loop()
{
 while (1)
 {
 / ... User code ... /
 zboss_main_loop_iteration();
 / ... User code ... /
 }
}
```

```
if (zboss_start() == RET_OK)
 {
 my_main_loop();
 }
else
 {
 /* zboss_start failed */
 }
```

Zigbee commissioning event handling

As the full-featured Zigbee stack is implemented, all Zigbee commands and events are processed. In addition, there are tools that allow the application to handle some Zigbee events, including network formation, association status, and leave indication. Due to its nature, different Zigbee events appear asynchronously and the stack informs the application with application signals. The application can handle or ignore these signals.

All application signals are processed in the application with a predefined callback function zboss_signal_handler(). Each application must implement this function, even if there is no need to process signals. The prototype for the signal handler function is:

```
void zboss_signal_handler(zb_uint8_t param);
```

In this function, `param` is a reference to a memory buffer.

Take into account the following policies while implementing the zboss_signal_handler() function:

- Signal processing must not perform long operations synchronously. As soon as the cooperative multitasking is implemented by the scheduler, each blocking operation blocks the entire Zigbee stack.
- If a valid reference to a memory buffer is passed to the application callback function, the application is responsible for managing this memory buffer. The memory buffer must be either reused or released by the application, otherwise the buffer is lost and memory leak occurs.

An application signal is described with the following parameters:

- Signal ID – see the zdo_app_signal_type section of the API Reference.
- Signal status – the API ZB_GET_APP_SIGNAL_STATUS() is used to get the status.
- Signal specific parameters – these are optionally provided for some signals; to see which parameters correspond to an event, see the description of the signals in the zdo_app_signal_type section in Zigbee stack API.

To get the signal description, the application calls zb_get_app_signal(). For example:

```
zb_zdo_app_signal_t zb_get_app_signal(zb_uint8_t param, zb_zdo_app_signal_hdr_t **p_sg_p);
```

Where:

- param is the reference to a memory buffer.
- ev_p is the pointer to store the extended event info; can be NULL. The function returns an ID of the application signal.

Example

zboss_signal_handler() implementation example:

```
/**@brief Zigbee stack event handler.

 * @param[in] param Reference to the ZigBee stack buffer used to pass arguments (signal).
 */
void zboss_signal_handler(zb_uint8_t param)
{
 zb_zdo_app_signal_hdr_t * p_sg_p = NULL;
 zb_zdo_signal_leave_params_t * p_leave_params = NULL;
 enum zb_zdo_app_signal_type_e sig = zb_get_app_signal(param, &p_sg_p);
 zb_ret_t status = ZB_GET_APP_SIGNAL_STATUS(param);

 switch(sig)
 {
 case ZB_BDB_SIGNAL_DEVICE_FIRST_START:
 case ZB_BDB_SIGNAL_DEVICE_REBOOT:
 if (status == RET_OK)
 {
 /* Joined network successfully */
 bdb_start_top_level_commissioning(ZB_BDB_NETWORK_STEERING);
 }
 else
 {
 /* Failed to join network. */
 }
 break;

 case ZB_ZDO_SIGNAL_LEAVE:
 if (status == RET_OK)
 {
 p_leave_params = ZB_ZDO_SIGNAL_GET_PARAMS(p_sg_p, zb_zdo_signal_leave_params_t);
 /* Network left. Handle leave parameters leave_params */
 }
 else
 {
 /* Unable to leave network */
 }
 break;

 case ZB_ZDO_SIGNAL_PRODUCTION_CONFIG_READY:
 if (status != RET_OK)
 {
 /* Production config is not present or invalid */
 }
 break;

 default:
 /* Unhandled signal. For more information see: zb_zdo_app_signal_type_e and zb_ret_e */
 break;
 }

 if (param)
 {
 zb_buf_free(param);
 }
}
```

BDB Commissioning API

The stack implements procedures for the following BDB commissioning modes:

- Network steering
- Network formation
- Finding and binding

The implementation can use commissioning at any time. For example, network steering can be started at any time for the whole node, and finding and binding can be performed at any time on any application endpoint.

Note

An explicit start of commissioning is required if zboss_start_no_autostart() was called instead of zboss_start(). For details, see Stack commissioning start sequence.

The commissioning procedure is initiated by triggering a top-level commissioning procedure with the bdb_start_top_level_commissioning function.

Finding and Binding

The finding and binding procedure can be initiated independently or within the commissioning process. To bind multiple endpoints on a target device with one endpoint on an initiator device, you must run different finding and binding procedures for all endpoint pairs consequently.

Note

The finding and binding procedure is implemented as a singleton, so there can only be one such procedure running at a time.

On the initiator device:

- After the network steering completion, call zb_bdb_finding_binding_initiator that sends the identify query.
- When you need to cancel the procedure, call zb_bdb_finding_binding_initiator_cancel.
- Receive the ZB_BDB_SIGNAL_FINDING_AND_BINDING_INITIATOR_FINISHED signal that is generated in ZB_NWK_BROADCAST_DELIVERY_TIME = 9 seconds.
- Repeat the procedure for every endpoint if needed.

On the target device:

- After the network steering completion, call zb_bdb_finding_binding_target, device shall respond the identify queries for `ZB_BDBC_MIN_COMMISSIONING_TIME_S` = 180 seconds.
- When you need to cancel the procedure, call zb_bdb_finding_binding_initiator_cancel.
- Receive the ZB_BDB_SIGNAL_FINDING_AND_BINDING_TARGET_FINISHED signal that is generated in `ZB_BDBC_MIN_COMMISSIONING_TIME_S` = 180 seconds.
- Repeat the procedure for every endpoint if needed.

APIs

The following table shows an overview of the commissioning API. For details, see BDB commissioning API in the Zigbee stack API documentation.

Macro	Parameters	Return value	Description
bdb_start_top_level_commissioning(zb_uint8_t mode_mask)	`zb_uint8_t` mode_mask – logical OR of values from zb_bdb_commissioning_mode_mask_t.	ZB_TRUE – in case the device starts successfully ZB_FALSE – in case an error occured (for example: the device has already been running)	Initiate BDB commissioning procedure defined by the `mode_mask`.
zb_bdb_finding_binding_target(zb_uint8_t endpoint)	`zb_uint8_t` endpoint – endpoint to bind.	RET_OK if procedure was successfully started	**Finding and binding interface.** Set finding and binding target mode on the endpoint.
zb_bdb_finding_binding_initiator(zb_uint8_t endpoint, zb_bdb_comm_binding_callback_t user_binding_cb)	`zb_uint8_t` endpoint – endpoint to bind. `user_binding_cb` - user callback function.	RET_OK, or the error code	**Finding and binding interface.** Initiate finding and binding on the endpoint. Pass result with `user_binding_cb`.
typedef zb_bool_t (* zb_bdb_comm_binding_callback_t)(zb_int16_t status, zb_ieee_addr_t addr, zb_uint8_t ep, zb_uint16_t cluster)	`status` – status (ask user, success or fail). `addr` – extended address of a device to bind ep – endpoint of a device to bind. `cluster` – cluster ID to bind.	zb_bool_t – agree or disagree to bind device.	**Finding and binding interface.** BDB finding and binding callback template.

Commissioning API calls

Note

Finding and binding (both target and initiator) must always be started explicitly.

Resetting to factory defaults

The Zigbee stack provides an interactive mechanism to reset any node implementation to the default factory settings, based on the requirements in the official Zigbee BDB specification. There are the following approaches for this process:

1. Resetting to factory defaults externally
2. Resetting to factory defaults with a local action

The resetting process of a Zigbee-based application includes persistent data cleanup and consists of the following steps:

1. Optionally, inform the user about the reset to the factory settings.
2. Cancel any rejoin process if it is in progress. (The action is performed by the stack.)
3. Stop any pending send or resend callbacks; free the buffers that are locked for these operations.
4. Do one of the following:
   - If the device supports OTA Upgrade process: stop this process by using zcl_ota_abort().
   - If the device supports Poll Control cluster: stop the check-ins by using zb_zcl_poll_control_stop().
   - If the device supports Power Config cluster: reset Power Config reporting to default.
5. Reset attributes for each supported cluster.
6. Flush new attribute values to a persistent storage.

> **Note**
>
> See zb_nvram_dataset_types_e for dataset type list. Many datasets are written by the stack internally, but the following ones must be written by the application:
> - ZB_NVRAM_HA_DATA
> - ZB_NVRAM_ZCL_REPORTING_DATA
> - ZB_NVRAM_HA_POLL_CONTROL_DATA (use zb_zcl_poll_control_save_nvram())

7. Broadcast NWK leave.
8. At NWK leave confirmation:
   - Optionally, inform the user about the completion of the factory reset process.
   - If needed, schedule device reboot.
9. Free resources that were reserved by the application.

> **Note**
>
> The application must not clean up the APS resources, for example in APS binding table, APS group table, and APS security keys storage. The stack performs the cleanup after the application sends the NWK Leave request to its own node.

Resetting to factory defaults externally

The stack provides a feature to reset a device externally by sending ZB_ZCL_BASIC_SEND_RESET_REQ. The target device goes through all the steps mentioned above in Resetting to factory defaults. The reset can be performed at any time once the device is started (see@ zboss_start()). The device initiates either ZB_ZCL_SET_DEFAULT_VALUE_CB in case it is predefined or the device user application callback with ZB_ZCL_BASIC_RESET_CB_ID.

Resetting to factory defaults with a local action

The device can be reset to factory defaults with a local action. For instance, such a reset can be initiated by the device button press or internal application logics. To do this, use zb_bdb_reset_via_local_action(zb_uint8_t param). The device performs NLME leave and cleans all Zigbee persistent data, except the outgoing NWK frame counter and application datasets (if any).

The application can reset its datasets before starting the process or after its completion. The reset can be performed at any time once the device is started (see@ zboss_start()).

After the reset, the application receives the ZB_ZDO_SIGNAL_LEAVE signal that notifies the user about the completion of the reset procedure.

Security

The application has access to a selection of security measures in the stack. However, most options are invisible to the application, but the features provided by Zigbee network security API and Zigbee API for code security installation can be used to set up the necessary security measures.

Zigbee network security API

Trust Center generates random network keys. This is the default behavior, according to **Zigbee PRO core specification <https://csa-iot.org/developer-resource/specifications-download-request/ >**.

It is possible to predefine the network key by using zb_secur_setup_nwk_key(). You can use this API call for debugging purposes to simplify the analysis of traffic sniffing results. Do not use it in the production environment.

An application acting as a coordinator (Trust Center) can force the network key switch procedure by calling zb_secur_setup_nwk_key().

Zigbee API for code security installation

In accordance with the Zigbee 3.0 specifications, this SDK provides install code functionality. The following table provides an overview of the related API macros.

Macro	Parameters	Return value	Description
void zb_secur_ic_add(zb_ieee_addr_t address, zb_uint8_t ic_type, zb_uint8_t *ic, zb_secur_ic_add_cb_t cb);	`zb_ieee_addr_t` address – long address of the device. `zb_uint8_t` ic_type - install code type as enumerated in zb_ic_types_t `zb_uint8_t` *ic – pointer to the install code buffer. `zb_secur_ic_add_cb_t` cb – callback that will be called after installcode addition.	Does not have return value.	Add the install code for the device with the specified long address. **Only for the coordinator.**
zb_ret_t zb_secur_ic_set(zb_uint8_t ic_type, zb_uint8_t *ic);	`zb_uint8_t` ic_type - install code type as enumerated in zb_ic_types_t `zb_uint8_t` *ic – pointer to the install code buffer.	`zb_ret_t` RET_OK on success. RET_CONVERSION_ERROR on error in install code CRC.	Add the install code for a device. **Not for the coordinator.**
void zb_secur_ic_remove_req(zb_uint8_t param);	`zb_uint8_t` param – ID of the buffer with request parameters, it will be also used to store response.	Does not have return value.	Remove the install code for the device with the specified long address. **Only for the coordinator.**
void zb_set_installcode_policy(zb_bool_t allow_ic_only);	`zb_bool_t` allow_ic_only – If True, Install code is required for adding a new device to the network. If False, the default commisioning is used.	n/a	Set Install code policy flag.

Install code API calls