life.augmented

AN5498

Application note

How to use Zigbee® clusters templates on STM32WB Series

Introduction

This document describes how to use Zigbee® clusters on STM32WB Series.

Zigbee applications are usually built on top of the Zigbee cluster library.

This application note indicates how to use these clusters, and how to control them.

Parts of this document are under Copyright © 2019-2020 Exegin Technologies Limited. Reproduced with permission.

1 General information

This document applies to the STM32WB Series dual-core Arm®-based microcontrollers.

Note: Arm is a registered trademark of Arm Limited (or its subsidiaries) in the US and/or elsewhere.

1.1 Reference documents

- [R1] 05-2374-22 ZigBee PRO Specification Revision 22
- [R2] 07-5123-07 Zigbee Cluster Library Revision 7
- [R3] AN5500 ZSDK API implementation for Zigbee® on STM32WB Series
- [R4] 07-5356-21 Zigbee Smart Energy Standard Version 1.4

1.2 Quick start

This document guides the developer through the essentials of building applications using the Exegin ZCL cluster templates. Each Exegin ZCL cluster template provides the starting source code for a complete cluster implementation. The template provides the mandatory ZCL commands and attributes as defined in the Exegin specification [R2], [R4].

Many commands and attributes have already successfully completed Zigbee® certification testing. Building an application consists of adding interfaces to specific hardware or similar application specific details. This is particularly true of application centric clusters whose main purpose is to provide application specific functionality such as the OnOff cluster, which in the case of a light turns that light on and off.

In addition to application specific clusters, templates are provided for other types of clusters such as support clusters and speciality clusters. Support clusters have minimal application interaction and either support general operation and/or support other clusters. The basic cluster supports general operation by providing information like the make and model of the device. The groups supports other clusters by enabling them to be group addressed. The third type of cluster, Specialty, are largely standalone and have minimal application involvement. These clusters provide some special function such as Touchlink, CBKE (Smart Energy security), and Green Power.

This document is intended to be used in conjunction with [R2], the ZCL 7 Specification which defines the clusters and [R3], for the Exegin ZSDK API.

This document builds on [R2] and [R3], provides to the user the knowledge to build applications using the cluster templates.

Section 2 Working with cluster templates contains the basic information about how the API works and Section 3 Application centric clusters the specific information on the cluster template.

Section 4 Support clusters contains the essential information on which to rely to build applications.

Section 5 Special dedicated endpoint clusters contains information about more specialized clusters.

Note: To build the applications is recommended to start with the checklist below and then review the subsequent sections.

1.3 Checklist

- Ensure the documents in the references section are available
- Assess which ZCL clusters the application requires
- See Section 3.1 Application centric client clusters to implement the client side of clusters
- See Section 3.2 Application centric server clusters to implement the server side of clusters
- See Section 4 Support clusters for basic, alarms, groups, scenes, alarms and identify clusters
- See Section 5 Special dedicated endpoint clusters for CBKE, touchlink and green power
- Applications requiring multiple cluster instances require multiple endpoints

The application is responsible :

- For instantiating endpoints and clusters
- For client clusters:
  - Sending command and attribute requests from local clusters to remote server cluster as needed by the application
  - In some rare cases, client clusters may support their own attributes
- For server clusters:
  - Either writing the local attribute when the value changes, or providing a callback that provides the value on demand
  - Providing callback functions to handle remote command requests and provide command responses
  - Interfacing with hardware

Most applications have other responsibilities such as supporting a user interface, controlling hardware, or other non-Zigbee related interfacing or services. All of these responsibilities are outside the scope of this document.

1.4 Callbacks and application structure

The cluster template functions utilize application-provided callbacks, which have a prototype d:

```
enum ZclStatusCodeT ZbFunction(...,
 void (*callback)(...,
void * arg), void *arg);
```

When the application calls this function it provides a callback function and an argument. The function returns quickly, and when the operation completes the callback is called. When the callback is invoked, it is passed back (to callback) the same argument that was provided when the original function (ZbFunction) was called.

Callbacks often need access to the ZSDK stack pointer and other application data. The application also needs somewhere to store pointers to clusters. A common convention defines an application storage structure to store this information.

```
struct application;
{
struct ZigbeeT * zb;
struct ZbZclClusterT * a_cluster;
/* other application data */
}
struct application app;
…
status = ZbFunction(..., callbk, &app)
…
void callbk(..., void *arg)
{
struct application *app = (struct application *)arg;
app ->zb ...
app->a_cluster ...
}
```

However, it is a good practice to only expose what is really needed in the callback. In some cases it is preferable to pass the cluster pointer. This also makes the callback more generic as the same callback, which can potentially be used with different cluster instances.

1.5 Clusters and endpoints

The stack is responsible for establishing and maintaining communication between a node and other nodes on a Zigbee network (see [R1] and [R3]). Application components build on these lower layers and exchange Zigbee cluster library (ZCL) messages.

This document describes how to build applications using the cluster templates provided with the ZSDK.

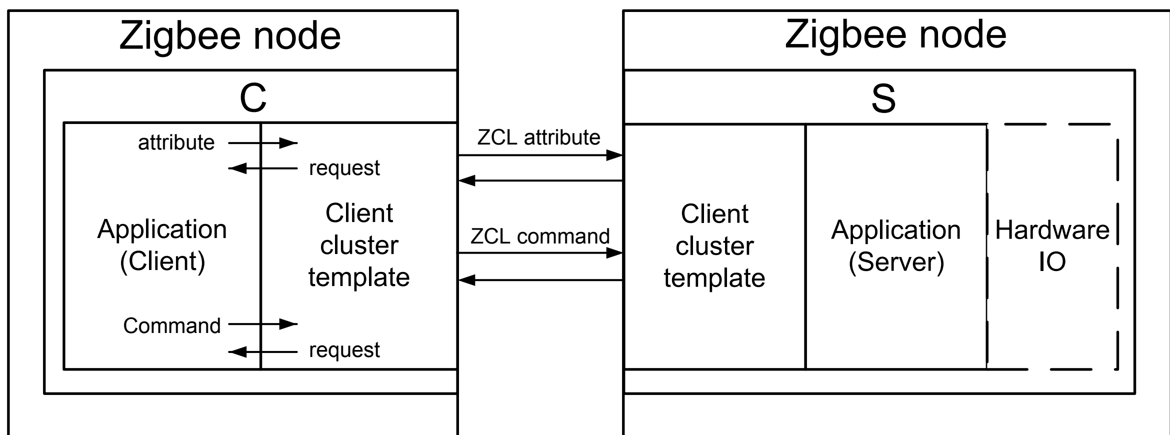
The ZCL is composed of organized sets of related functionality called clusters. Often this functionality is associated with a specific hardware element such as an individual light, or a switch that controls the light. All the functionality associated with a given device resides on a single endpoint.

Each endpoint on a node has a unique endpoint id (usually referred to as just “endpoint”), ranging from 1 to 239. Endpoints 0 and 240 to 255 are reserved for special uses.

1.6 Client server relationship

The cluster functionality is organized into a server side, which provides a service on an endpoint of one node, and a client side, which accesses that service via another endpoint on another node.

Figure 1. **Client server architecture**



A node may have multiple physical devices (for instance, it supports multiple lights or multiple switches) by supporting multiple instances of the same cluster (for instance OnOff cluster). Since that, each instance resides on a unique endpoint, a specific light (OnOff server) on one node is associated with a specific switch (OnOff client) on another node by the use of the respective node endpoints.

For example: A switch 3 may reside on endpoint 3 on a switch node and be configured to communicate with light 2 on endpoint 2 of a light node.

Only a single instance of a cluster is allowed on a given endpoint, but each endpoint typically has multiple clusters (for instance OnOff, Basic, Alarms, Scenes, etc.).

An application creates one or more endpoints using the `ZbZclAddEndpoint()` function, which is declared in the Zigbee cluster library header file, zcl.h.

Note: It is also possible to create an endpoint using an APS function.
For ZCL applications the above API is preferred.

When creating a new endpoint, the application should select a new unique endpoint between 1 and 239, and provide the ProfileId and DeviceID.

In the following example we are choosing endpoint 1. Values for ProfileId and DeviceId begin with ZCL_PROFILE_ and ZCL_DEVICE_ and are defined in the header zcl.h.

```
ZbApsmeAddEndpointReqT add_ep_req;
ZbApsmeAddEndpointConfT add_ep_conf;
memset(&add_ep_req, 0, sizeof(add_ep_req));
memset(&add_ep_conf, 0, sizeof(add_ep_conf));
add_ep_req.endpoint = 1;
add_ep_req.profileId = ZCL_PROFILE_HOME_AUTOMATION;
add_ep_req.deviceId = ZCL_DEVICE_ONOFF_SWITCH;
ZbZclAddEndpoint(app->zb, &add_ep_req, &add_ep_conf);
```

After an endpoint is created, there are ZCL helper functions such as ZbZclClusterSetProfileId() that help with changing endpoint settings.

Note: It is always recommended to zero data structures using memset before to use to prevent unpredictable behavior caused by uninitialized memory.

Most stack functions take the stack pointer (struct ZigbeeT pointer as their first argument. Most ZCL functions take ZbZclClusterT as their first argument (some take ZigbeeT instead).

The prototype of ZbZclAddEndpoint() is common, after the stack.

The add endpoint request and confirmation are defined in the header file Zigbee.aps.h.

1.7 Endpoints and simple descriptors

When the user creates an endpoint, the stack internally creates a simple descriptor (see [R1] section 2.3.2.5) for each endpoint created. The simple descriptor has two cluster lists: “input” and “output”. ZCL server clusters reside on the input list, and the client clusters reside on the output list.

The ZDO Simple_Desc_req can be used to query the simple descriptor of a remote device. However, in practice the Match_Desc_req ZDO command is more commonly used to locate clusters on the remote node. When using Match_Desc_req and Simple_Desc_req, it may be useful to use Active_EP_req to discover a remote node’s endpoints first.

See [R1] and [R3] for more information on these ZDO commands.

1.8 Types of clusters

After creating an endpoint, the application can create application clusters and add them to that endpoint. By default, every endpoint includes the basic cluster, which will be discussed in more detail later in this document.

In addition to the basic cluster, there are several other special clusters including: groups, scenes, touchlink, CBKE, alarms, and green power.

1.9 Application centric, support, and special clusters

Zigbee applications are built using clusters. Most clusters provide application functionality such as: OnOff, color control, temperature measurement, smart energy metering, etc. A Zigbee node provides endpoints, which each contain an instance of one or more application centric clusters.

There are also support clusters such as groups, scenes and alarms, which do not provide such application functionality. Rather these support clusters expose functionality that supports other clusters, enabling group addressing of other cluster, activation of settings for other clusters and logging of alarms generated by other clusters, respectively.

Additionally, there are some special clusters, which are automatically created by the stack, to provide special functionality not directly related to the application. These clusters, such as CBKE, touchlink, and green power proxy basic, also often reside on their own endpoint.

The following sections cover the application centric clusters first, followed by the support clusters and then special clusters.

1.10 Commands: cluster specific versus profile wide

ZCL clusters provide a set of attributes and commands (some clusters provide only attributes). Commands are divided into two types: cluster specific (specific to that cluster such as basic or OnOff) and profile wide (available on every cluster). The main purpose of profile-wide commands is accessing a cluster's attributes. Each cluster has its own unique cluster specific commands. Attributes are primarily accessed through profile-wide commands (read attributes, write attribute), but the profile-wide commands also include other general functions such as the default response, configure reporting, discover attributes and commands, etc. The profile-wide commands are the same for all clusters. Profile Wide commands are described in Section 2.5 General Command Frames of the ZCL 7 [R2].

1.11 Addressing and bindings

There are several mechanisms available for an application to address an attribute or command request from the source (normally client) cluster to the destination (normally server) cluster. Network short addresses are used expensively in Zigbee networks so unicast addressing using short address is a natural method. The extended address can also be used, but this is uncommon. Group addressing can be used to multicast several devices with a single request when groups are used (see Section 4.2 Groups cluster). However, for practical applications most Zigbee applications use bindings.

Bindings are pre-configured source-to-destination addresses. They can be configured locally on a node by the application or remotely, for example by a separate commissioning tool, using over the air ZDO messages. For a specific cluster (Cluster ID), a binding associates a local endpoint with a remote network short address and endpoint. Applications use bindings by selecting the source cluster and using ZB_APSDE_ADDRMODE_NOTPRESENT as the destination. The APS layer then automatically scans the binding table for bindings with a matching source endpoint and Cluster ID. The request is then sent to all bindings that match. Only a single response is returned to the application, regardless of the number of bindings.

The use of bindings makes the application simpler by removing the burden of determining the appropriate addresses to send a message at every location where a message is to be sent and places it in a single central commissioning step. For some applications, the Find-and-Bind mechanism can be good technique to establish these bindings. The application can also discover clusters on remote endpoints using standard ZDO services (such as `ZbZdoMatchDescReq()` and `ZbZdoSimpleDescReq()`) and create its own local bindings using `ZbApsmeBindReq()` or in some cases send bindings to the remote devices (usually back to the local node) using `ZbZdoBindReq()`.

The struct `ZbApsAddrT` is widely used in addressing. Rather than having to declare a `ZbApsAddrT` every time the addressing is to binding, the helper ZbApsAddrBinding can be specified in order to use the available bindings.

For client applications bindings are commonly used to address command and attribute requests. On a server it is possible to configure reporting of changes in attribute values. On server clusters bindings determine the destination of reports.

A client that needs to receive reports, used `ZbZdoBindReq()` to create a binding on the server back to itself, then creates a Report Configuration using `ZbZclAttrReportConfigReq()` to configure the report. Then when the server needs to send a report it uses the local server-side binding created by the client to know where to send the report.

1.12 Find and bind

The BDB provides a semi-automated method to create bindings, built upon the Identify cluster's Identify mode. The finding and binding process can be initiated automatically after joining (for instance when `ZbStartup()` is called with `ZbStartTypeJoin`). Automatic find and bind occurs when `BDB_COMMISSION_MODE_FIND_BIND` is enabled in `ZbStartupTbdbComissioningMode`. The application can also initiate finding and binding manually at any time by calling `ZbStartupFindBindStart()`.

The find and bind relies upon target supporting the identify server cluster. Prior to the initiator performing find and bind, the target or targets must enter identify mode. They enter Identify mode, either by some local action such as a button press or by an Identify command from the initiator or some other device on the network such as a commissioning tool.

The find and bind initiator sends a broadcast IdentifyQuery command (`zcl_identify_query_request()`). Targets in identify mode respond with a nonzero Identify time. The initiator then requests the Simple Descriptor (using `ZbZdoSimpleDescReq()`) from each node in identify mode. The initiator then creates a local binding for each local client cluster with a matching remote server cluster. An important note is that this process relies on the client clusters being instantiated on the initiator prior to find and bind.

The find and bind process has several limitations. It can only bind local client clusters to remote server clusters. Usually this is the desired behavior; however, there are cases (such as the smart energy messaging cluster) where the opposite server to client bindings are needed.

Also finding and binding indiscriminately binds initiator client endpoints with target endpoints. This is fine when there is one client endpoint and one server endpoint. In applications where there is more than one client endpoint and/or more than one server endpoint this is not the desirable approach.

Finding and binding does not solve all application commissioning needs. The application can always establish its own bindings by reusing the same mechanisms (for instance ZDO commands) as find and binding using them in different application specific ways to create the needed bindings.

2 Working with cluster templates

2.1 The cluster pointer - ZbZclClusterT

All clusters are represented by the same `ZbZclClusterT` datatype, which represent an instance of a specific cluster. In general, the internal structure of this datatype is of no use to the application; however, it is very important because it is used throughout the cluster APIs as the generic representation of any cluster, regardless of the type of cluster. Because of this many APIs are generic and works with any cluster. For example `ZbZclReadReq()` allows you to read the attribute in any cluster, whereas `ZbZclDoorLockClientLockReq()` only works with a Door Lock client cluster handle, but both use the same `ZbZclClusterT` datatype. In general, the distinction is by the context.

2.2 Cluster naming conventions

There is also a naming convention requiring that cluster specific APIs include the name of the cluster (no cluster in the case of `ZbZclReadReq()` and `ZbZclDoorLockClientLockReq()` for the Door Lock client cluster). The defining header file is also an indication, `ZbZclReadReq()` is defined in the general “zcl.h” whereas `ZbZclDoorLockClientLockReq()` is defined in the cluster specific header “zcl.doorlock.h”. Finally, the cluster specific APIs return `ZCL_STATUS_FAILURE` if a cluster handle of the wrong type is provided. It is important to always check the status code to prevent unexpected behavior.

2.3 Creating a new instance of a cluster

```
#include "zcl.x.h"
app->x_client_cluster = ZbZclXClientAlloc(zb, endpoint, …)
or
app->x_server_cluster = ZbZclXServerAlloc(zb, endpoint, …)
if (app->x_server_cluster == NULL) {
/* handle error */
```

The allocation functions return NULL on error, otherwise a cluster handle is returned. In general, the application never examines the contents of this struct. Instead this handle is used in most cluster library applications.

Like most ZSDK API functions the allocation functions take the `structZigbeeT` * stack pointer as their first argument. This binds the new cluster instance to the stack instance. From this point on ZCL API functions take the `structZbZclClusterT` * returned by the allocation function, the reference to the newly created cluster instance.

The second argument to allocation functions is the endpoint id. This binds the newly created cluster to that endpoint. Multiple cluster instances can be bound to the same endpoint, provided there is only one instance of any given cluster.

After the endpoint, the remainder of the arguments are specific to the cluster.

Server clusters with multiple commands usually take a structure with multiple callbacks, one for each command that the application supports. The application provides a callback for each command that it supports. If the application provides NULL for the entire structure pointer or a specific command callback, the cluster responds with a Default Response of `ZCL_STATUS_UNSUPP_CLUSTER_COMMAND` for the specific command (or every command if the entire structure pointer is `NULL`).

Here is an example of how the application would implement such a callback, providing function `app_get_profile_info()` which is called whenever the `ZCL_ELEC_MEAS_CLI_GET_PROFILE_INFO` command is received.

```
enum ZclStatusCodeT app_get_profile_info (struct ZbZclClusterT *cluster,
structZbZclAddrInfoT *src_info, void *arg)
{
/* TODO handle get profile info command */
return ZCL_STATUS_SUCCESS;
}
…
struct zcl_elec_meas_callbacks_t callbacks = {
app_get_profile_info,
NULL,
};
…
cluster = ZbZclElecMeasServerAlloc (zb, endpoint, callbacks, app);
…
```

Note: *In this example, the Get Measurement Profile callback is declared NULL. When this command is received, with a Default Response of* `ZCL_STATUS_UNSUPP_CLUSTER_COMMAND` *is automatically sent.*

The enum ZclStatusCodeT defines the available status codes.

2.4 Remote versus local clusters

One aspect that is sometimes confusing when working with clusters is the distinction whether the cluster is local or remote. Part of this confusion comes from the distinction between client and server clusters. Server clusters reside on the network node that provides functionality and this server functionality is accessed from a remote network node through its local client cluster. Consider an attribute which resides on the server cluster. The following example demonstrates how a remote client node sends a write attributes request.

2.5 Remote write attribute

```
#include "zcl.h"
ZbZclWriteReqT *req;
/* populate req */
memset(&req, 0, sizeof(req));
req.dst.mode = ZB_APSDE_ADDRMODE_NOTPRESENT; /* send to binding */
req.count = 1;
req.attr[0].attrId = ZCL_BASIC_ATTR_ENVIRONMENT;
req.attr[0].type = ZCL_DATATYPE_ENUMERATION_8BIT;
req.attr[0].value = ZCL_BASIC_ENVIRONMENT_MIRROR_SUPPORT;
req.attr[0].length = 1;
status = ZbZclWriteReq (app->cluster, req, read_rsp_callback, &arg);
if (status != ZCL_STATUS_SUCCESS) {
/* handle error */
```

Note: *the client accesses the remote attribute using ZbZclWriteReq() . In general, over-the-air messages have request (or Req) in their name.*

However, the attribute is local to the server and the server must also be able to write to it to update the value, which will be read by remote nodes.

2.6 Local write attribute

```
#include "zcl.h"
status = ZbZclAttrIntegerWrite(app->server_cluster, ZCL_BASIC_ATTR_ENVIRONMENT, 0x03);
/* ... */
```

As defined in the Zigbee spec, some attributes are read-write and others are read-only. Read-only means that they cannot be written over-the-air, for instance remotely. They must be writable locally by the application. It is important to note that on the server the local application can always write to an attribute, even attributes that are read-only over-the-air.

2.7 Default PICS settings

When clusters are certified by a Zigbee Alliance authorized test house, a PICS (Protocol implementation conformance statement) is required for each cluster certified.

The PICS includes a checklist of implemented features including attributes and commands. The provided cluster templates include the PICS settings for the cluster as delivered.

The following code is an example from the scenes cluster:

```
/* PICS.ZCL.Scenes
 *
 * S.S | True
 * S.C | True
 *
 * Server Attributes
 * S.S.A0000 | True
 * S.S.A0001 | True
 * S.S.A0002 | True
…
 * Commands Received
 * S.S.C00.Rsp | True
…
```

The PICS comments are provided only for clusters for which there is a published PICS document such as of ZCL 7, and can be used as a starting point to certify the functionality. However, as a specific implementation may choose to add optional commands and attributes to the cluster, this needs to be updated by changing the status of individual items.

2.8 Entering PICS Information Into ZTT

When running test cases using the ZTT (Zigbee Test Tool), the information contained in the PICS comments in the cluster's header file must be entered manually into the ZTT.

This entry is accomplished by:

1. Opening the ZTT
2. Navigating to the PICS tab
3. Scrolling down to find the appropriate cluster
4. Selecting implemented features (features listed as TRUE in the PICS comments)

The ZTT displays the supported test cases and allow users to run them.

3 Application centric clusters

The ZCL includes many application specific clusters reflecting the diversity of possible supported devices. The application specific clusters define the device and its capabilities. Examples include a thermostat cluster on a thermostat, window covering cluster on a blind, or meter cluster on a smart energy meter. Other application specific clusters are related to specific functionality that may or may not be present on particular device. Advanced thermostats may support the thermostat user interface configuration cluster whereas simple thermostats may not. As another example, a HVAC controller may support the pump configuration and control, level control and OnOff clusters in order to support its functionality.

The common element in application specific clusters is that they support a device specific use case, for instance operation as a thermostat, HVAC controller, smart energy controller or blind. In most cases the application specific clusters provide an interface to hardware. This compares to support clusters such as the basic cluster, groups, scenes, alarms and identify which provide general functionality in support of the other clusters or Special clusters such as Touchlink, CBKE and green power, providing capabilities unrelated to the device’s main purpose.

The application specific clusters are delivered as templates that must be completed by the application. The OnOff cluster template handles the details of attribute and command message handling, but the application must complete the details of now an On command translates to setting a GPIO, for example.

All application specific clusters follow a common pattern. This section shows this pattern and how it is used in the application code.

Table 1. STM32WB Zigbee clusters

Cluster	Header file	Cluster	Header file
Power configuration	zcl.power.config.h	Device temperature configuration	zcl.device.temp.h
OnOff	zcl.onoff.h	On/Off switch configuration	zcl.onoff.swconfig.h
Level control	zcl.level.h	Time	zcl.time.h
RSSI location	zcl.rssi.loc.h	Commissioning cluster	Zcl.commission.h
OTA update cluster	zcl.ota.h	Power profile	Zcl.power.profile.h
Poll Control	zcl.poll.control.h	Nearest gateway	zcl.nearest.gw.h
Door lock	zcl.doorlock.h	Window Covering	zcl.window.h
Pump configuration and control	zcl.pump.h	Thermostat	zcl.therm.h
Fan control	zcl.fan.h	Dehumidification control	zcl.dehum.ctrl.h
Thermostat UI configuration	zcl.therm.ui.h	Color control	zcl.color.h
Ballast configuration	zcl.ballast.config.h	Illuminance measurement	zcl.illum.meas.h
Illuminance level sensing	zcl.illum.level.h	Temperature measurement	zcl.temp.meas.h
Pressure measurement	zcl.press.meas.h	Diagnostics	zcl.diagnostics.h
Occupancy sensing	zcl.occupancy.h	IAS zone	zcl.ias_zone.h
IAS ancillary control equipment	zcl.ias_ace.h	IAS warning device	zcl.ias_wd.h
Voice over Zigbee	zcl.voice.h	Meter identification	zcl.meter.id.h
Electrical measurement	zcl.elec.meas.h	-	-

Note: Additional clusters may have been added after this list was compiled.

3.1 Application centric client clusters

Section 2.3 Creating a new instance of a cluster covers the creating client cluster instances. It is the responsibility of the application to save the cluster pointer (a recommended approach is to use an application storage structure that can be passed as the “arg” in callback functions). The pointer to the cluster instance is then used whenever the application wants to use the local client cluster to access the server on the remote node. Every application centric client cluster provides a common API. This section describes that common API. For the cluster specific details, such as the attribute names or command function, please refer to the header file of the specific cluster (for example refer to zcl.onoff.h for the OnOff cluster). The ZCL cluster headers include both the client and server APIs. The attributes and commands are defined in ZCL 7 [R2] except for smart energy clusters that are defined the ZSE Specification [R4].

3.1.1 Accessing remote server attributes from client clusters

The attributes on the remote server are accessed via a local client clusters.

Note: *In some cases, there are attributes on client clusters that are accessed from server clusters. For most clusters the only client attribute is the cluster revision, ZCL_GLOBAL_ATTR_CLUSTER_REV which must be present on every client and server cluster. Some client clusters have attributes, for instance, all cluster specific OTA and DRLC cluster attributes reside on the client. The metering cluster has mostly server attributes an a few client attributes (in addition to cluster revision) and the device management had attributes on both server and client sides.*

The attributes may be R (read-only), W (write-only), or RW (read-write) and additionally some are P (reportable). Each cluster header file includes an anonymous enum that defines the attribute IDs for that cluster, of the form ZCL_<cluster>_ATTR_<name> for example `ZCL_OCC_ATTR_SENSORTYPE` for the sensortype attribute in occupancy cluster. The following example reads the remote value of the SensorType attribute:

```
#include “zcl.occupancy.h”
void handle_read_rsp(const ZbZclReadRspT *rsp, void *arg)
{
struct application *app = (struct application *)arg;
if (rsp->status == ZCL_STATUS_SUCCESS) {
app->remote_occupancy = (uint8_t)*rsp.attr[0].value;
}
}
…
enum ZclStatusCodeT status;
ZbZclReadReqT req;
memset(&req, 0, sizeof(req));
req.dst = &ZbApsAddrBinding;
req.count = 1;
req.attr[0] = ZCL_OCC_ATTR_SENSORTYPE;
status = ZbZclReadReq(app->occ_client, req, handle_read_rsp, app);
if (status != ZCL_STATUS_SUCCESS) {
/* handle error */
…
```

Note: *The destination is “to binding”, which assumes that there is already a binding established from the client to a remote server.*

3.1.2 Sending commands to remote nodes

In general clients send command requests to servers, however as with attributes there are occasional exceptions. Client cluster templates provide functions that allow applications to send command requests to servers. The syntax of these commands is:

ZbZcl[Cluster][Client/Server]Command[Name]Req()

For instance ZbZclMeterClientCommandGetProfileReq() sends a Get Profile command request from the Meter cluster client. Many commands have a payload as defined in the ZCL or ZSE specification. The following table shows the meter cluster Get Profile command (See [R2] document):

Table 2. Get profile command payload

Octets	1	4	1
Data type	8-bit enumeration[(1)]	UTCTime	Unsigned 8-bit integer
Field name	Interval channel	End time	NumberOfPeriods

1. *CCB 1077*

The cluster template provides a C struct for every command payload. In this case ZbZclMeterClientGetProfileReqT struct is defined in “zcl.meter.h”. The application declares one of these structures and passes it to the command request.

The following code snippet illustrates the general process of sending a command from client to server and receiving a response.

```
struct application app;
...
void get_profile_callback(struct ZbZclCommandRspT * rsp, void *arg)
{
struct application *app = (struct application *)arg;
if (rsp.status == ZCL_STATUS_TIMEOUT) {
/* server did not respond within timeout */
}
else if(((rsp.hdr.frameCtrl.frameType & ZCL_FRAMETYPE_CLUSTER)==0)&&
(hdr.cmdId == ZCL_COMMAND_DEFAULT_RESPONSE)) {
/* server returned a default response, indicating error */
 }
else if(rsp.status == ZCL_STATUS_SUCCESS) {
 /* a valid response was returned */
}
else {
/* an error occurred */
}
}
...
enum ZclStatusCodeT status;
struct ZbZclMeterClientGetProfileReqT req;
memset(&req, 0, sizeof(req));
interval_channel = ZCL_METER_SAMPLE_TYPE_CONSUMP_DELIV;
end_time = 0; /* zero gets the most recent */
number_of_periods = 1;
status = ZbZclMeterClientCommandGetProfileReq(app.cal_cluster, & ZbApsAddrBinding, & req,
get_profile_callback, &app);
```

The application callback must handle four possible scenarios:

1. The ZCL response is received as expected
2. The server did not respond within the required amount of time. This could be caused when neither a NWK nor APS ack has occurred or a valid ZCL response (Default Response or Cluster Command specific response)
3. A Default Response was received indicating an error condition
4. A ZCL error status was returned

3.1.3 How clients receive reports from remote server clusters

As described in Section 1.11 Addressing and bindings, a client configures a remote server to send it reports by sending it a Binding to create a binding back to itself using ZbZdoBindReq() and creating a report configuration in the server cluster using ZbZclAttrReportConfigReq(). However, prior to performing this configuration on the server, the client should configure a local application callback to receive the reports. For instance:

```
void receive_report (struct ZbZclClusterT *cluster, ZbApsdeDataIndT *data,
uint16_t id, enum ZclDataTypeT type, const uint8_t *payload, uint16_t len)
{
 /* handle report */
}
...
struct ZbZclClusterT * client_cluster;
ZbZclClusterReportCallbackAttach(client_cluster, receive_report);
```

Whenever the server determines that it is time to send a report based on the report configuration, it uses the binding that the client has created in the server's binding table to send the report back to the client. When the client cluster receives the report, with the callback configured as shown above, this callback is invoked with the received report. The application can then process the report as needed.

3.2 Application centric server clusters

The previous Section 3.1 Application centric client clusters covers information pertaining to the application requirements for client clusters. Moreover, this section covers the server side.

The server clusters provide services for the use of other nodes on the network, exposing them via their local server cluster. The server cluster templates are shipped incomplete, for this reason, the functionality must be completed by the application. For instance, the OnOff server cluster template handles the mechanics of the ZCL messaging but the details of how the hardware is physically turned on or off must be implemented in the application. This section describes how the application completes the implementation of application centric server clusters.

Similar to client clusters, applications must create server clusters as described in Section 2.3 Creating a new instance of a cluster and maintain server cluster pointers. However, server cluster pointers are needed by the application less frequently than client cluster pointers.

3.2.1 Adding optional attributes to a cluster

The server cluster templates only create the mandatory attributes by default. If the application needs to define additional optional attributes, it can easily do so after creating the server cluster instance by calling `ZbZclAttrAppendList()` with the new optional attributes.

```
static const struct ZbZclAttrT optional_attr_list[] = {
{
ZCL_METER_SVR_ATTR_CURSUM_RECV, ZCL_DATATYPE_UNSIGNED_48BIT,
ZCL_ATTR_FLAG_NONE, 0, NULL,
{0, 0}, {0, 0}
},
{
ZCL_METER_SVR_ATTR_MAX_DMND_RECV, ZCL_DATATYPE_UNSIGNED_48BIT,
ZCL_ATTR_FLAG_NONE, 0, NULL,
{0, 0}, {0, 0}
},
…
};
…
app->meter_server = ZbZclMeterServerAlloc(zb, endpoint, NULL, &app);
if (ZbZclAttrAppendList(&app->meter_server, optional_attr_list,
ZCL_ATTR_LIST_LEN(optional_attr_list)) != ZCL_STATUS_SUCCESS) {
/* handle error */
}
…
```

In addition to adding new attributes, you can also override the built-in definitions of the mandatory attributes.

3.2.2 Updating attribute values via direct write

There are two approaches to updating the value of attributes. By default, the value of an attribute is maintained within the cluster itself. The server application can update this local value using a set of helper functions such as ZbZclIntegerWrite() for numeric attributes and ZbclAttrStringWriteShort() / ZbclAttrStringWriteLong() for string attributes. For example when the application decides it is time to update the value of the attribute in the previous example it can call:

```
long long max_demand_received;
ZbZclAttrIntegerWrite(&app->meter_server, ZCL_METER_SVR_ATTR_MAX_DMND_RECV,
max_demand_received);
```

Since the value is stored in the cluster, there may cases where the application needs the attribute, especially if it may be written by a remote client. The local attribute value can be read at any time using the helper `ZbZclAttrIntegerRead()`. These datatype specific helpers are for the most common data types. The general APIs to access any local cluster server attribute are `ZbZclAttrRead()` and `ZbZclAttrWrite()`. These general APIs are less convenient to use because attribute values are stored in over-the-air form which is generally little endian. For ZCL datatypes with an endianness, the `putle()` functions (from `pletoh.h`) should be used to convert from host to little endian form before use with `ZbZclAttrWrite()`; and `pletoh()` functions to convert read values using `ZbZclAttrRead()` to host endianness prior to use in the application.

This push method is one way that a server can maintain its attribute values. It is also possible to use callbacks as described in the next section.

3.2.3 Using cluster attribute callbacks

Instead of the application being control of the attribute value, the application can provide a callback function to perform the read (or write) when the attribute is accessed. This is particularly useful when the attribute value is determined by physical entity in hardware (such as a GPIO). Here is an example of providing a read callback that accesses an external memory mapped value represented in hardware:

```
static const struct ZbZclAttrT optional_attr_list[] =
{
{
ZCL_METER_SVR_ATTR_CURSUM_RECV, ZCL_DATATYPE_UNSIGNED_48BIT,
ZCL_ATTR_FLAG_CB_READ, 0, receive_callback,
{0, 0}, {0, 0}
},
#include "pletoh.h"volatile extern uint64_t current_sum_rcv;
enum ZclStatusCodeT receive_callback(struct ZbZclClusterT *cluster, struct ZbZclAttrCbInfoT
*info)
{switch(info->type)
{case ZCL_ATTR_CB_TYPE_READ:putle48(info->zcl_data, current_sum_rcv);
return ZCL_STATUS_SUCCESS;
default : return ZCL_STATUS_FAILURE;
}
return ZCL_STATUS_SUCCESS;
}
```

The callback attribute flag values for the callback are`ZCL_ATTR_FLAG_CB_READ`

`ZCL_ATTR_FLAG_CB_WRITE` (ZCL_ATTR_FLAG_CB_NOTIFY is a special case of write when written locally as opposed to over the air).

3.3 Implementing server-side command handlers

When clusters support commands (as mentioned previously this is generally but not strictly exclusive to the server side), the application is responsible to pass structure of callback functions to the cluster instantiation “Alloc” function.

```
static enum ZclStatusCodeT restart_device(struct ZbZclClusterT *clusterPtr,
struct ZbZclCommissionClientRestartDev *req,
structZbZclAddrInfoT *srcInfo, void *arg)
{
if ((req.options & ZCL_COMMISSION_OPTIONS_IMMEDIATE) != 0) {
 /* reset immediately */
 }
else {
...
return ZCL_STATUS_SUCCESS;
}
struct ZbZclCommissionServerCallbacksT callbacks =
{
restart_device,
NULL,
NULL,
NULL,
}
...
app->commission_server = ZbZclCommissionServerAlloc(zb, profile, aps_secured,
&callbacks, app);
```

In this example the server application implements only the restart device (ZCL_COMMISSION_CLI_CMD_RESTART_DEVICE) ZCL command. The payload of the Restart Device command is unpacked and provided to the callback as a C structure. The Save Startup, Restore Startup and Reset Startup callbacks are undefined. If these commands are received by this cluster a Default Response with status ZCL_STATUS_UNSUPP_CLUSTER_COMMAND is sent back client.

If the command callback returns anything other than ZCL_STATUS_SUCCESS a default response is sent, unless the client has requested that a Default Response be sent in the case of success. The callback can override this behavior by returning ZCL_STATUS_SUCCESS_NO_DEFAULT_RESPONSE which must only be done when the callback returns ZCL response message.

In the following example, the server request callback located the requested information, forms the structure for the response information (rsp) and used the server cluster ZbZclMeterServerSendGetProfileRsp() to send a ZCL response back to the source client.

```
static enum ZclStatusCodeT
meter_get_profile(struct ZbZclClusterT *clusterPtr,
void *arg,
struct ZbZclMeterClientGetProfileReqT *req,
struct ZbZclAddrInfoT * srcInfo)
{ZbZclMeterServerGetProfileRspT rsp;
/*
Obtain Profile info matching request */
...
memset(&rsp, 0, sizeof(rsp));
rsp.end_time = req ->end_time;
rsp.status = ZCL_METER_PRFL_STATUS_SUCCESS;
rsp.profile_interval_period = profile[i].profile_interval_period;
rsp.number_of_periods = profile[i].number_of_periods;
rsp.profile_data = profile_data;
rsp.profile_length = ZbZclMeterFormSampledData(profile_data, sizeof(profile_data),samples,
n);
if (rsp.profile_length < 0) {return ZCL_STATUS_INSUFFICIENT_SPACE;
}
ZbZclMeterServerSendGetProfileRsp(clusterPtr, srcInfo, &rsp);
return ZCL_STATUS_SUCCESS;}
```

Note: *How it returns ZCL_STATUS_SUCCESS informing the stack that it has handled the response. An error status results in a default response with the provided ZCL_STATUS_ error value.*

In a few cases client clusters support commands. In these rare cases all of the proceeding applies, the roles of client and server are simply reversed.

4 Support clusters

4.1 The basic cluster

The basic cluster is special in that a server instance is automatically created and added to every endpoint by the stack. Nodes can access the server cluster of a remote node by declaring a client basic cluster on one of their endpoints.

Unlike other clusters, the Basic server cluster it is a singleton cluster, i.e. there is only one instance of the Basic server cluster and it appears on every endpoint. This is because the information in the Basic cluster (such as the manufacturer and model) applies globally to the entire node. In order to access the functionality of the basic cluster, a special helper function is provided, ZbZclBasicWriteDirect(), see the header file “Zigbee.h”. This function take any valid endpoint on the device and details of the attribute you want to change. The updated attribute value is reflected in the basic cluster on all existing and future endpoints.

```
char * mfr_str = “Zigbee”;
uint8_t zb_mfr_str[7];
/* convert C string to Zigbee string */
memcpy(& zb_mfr_str[1], mfr_str, strlen(mfr_str));
zb_mfr_str[0] = strlen(mfr_str);
ZbZclBasicWriteDirect (app-> zb, ep, ZCL_BASIC_ATTR_MFR_NAME, zb_mfr_str, 7);
```

The basic cluster also supports two alarms: general hardware fault and general software fault. The generation of these alarms is controlled by the AlarmMask attribute. The local application may post one of these alarms by locally calling ZbZclBasicPostAlarm(), however the corresponding bit of the AlarmMask attribute may prevent it from being processed.

The basic server supports one command, ZCL_BASIC_RESET_FACTORY. Applications must register a callback with the stack, which is invoked if their node receives one of these reset to factory defaults. When this callback is received, it is the responsibility of the application to reset the values of all attributes in all clusters to their factory default values.

4.2 Groups cluster

The groups cluster provides a mechanism to manage group management of an endpoint. By adding the groups cluster to an endpoint, groups for that endpoint can be managed remotely via Group Cluster commands. Groups are a feature of APS stack layer and groups are stored internally in the Groups Table. Applications access group on their local endpoints using APS APIs such as `ZbApsmeAddGroupReq()` and `ZbApsmeRemoveGroupReq()`. The Groups server cluster allows remote applications (via the groups client cluster) to manage the group membership of an endpoint in the same way the local application does using the APS API. The groups cluster provides commands such as Add, Remove, Remove All, View, Get Membership, and Add Group If Identifying (which is linked to the ZCL Identify Cluster). See section 3.6 Groups [R2] and the header file `zcl.groups.h` for more details.

4.3 Scenes cluster

The scenes cluster provides the ability to apply a block of settings to another cluster or clusters on the same endpoint as Scenes Server cluster. Only a few select clusters support scenes. In the cluster specification, these clusters have an additional section with the title “Scene Table Extensions” OnOff see section 3.8.2.6 [R2] for the OnOff cluster for Color see section 5.2.2.5, Thermostat section 6.3.2.6, and Window Covering section 7.4.2.4 . This scene table extension section defines a set of specific attribute values, to be applied to attributes in that cluster. For example, the Window Covering Scene Extension Table consists of values of the CurrentPositionLiftPercentage and CurrentPositionTiltPercentage attributes in that order. The scenes cluster stores this data in the Extension Field (for example Figure 3-19). To the scenes cluster the Extension Field is an opaque blob of data that gets applied to a specific cluster that understands and interprets it.

When a scene is activated, specific Extension Fields are applied to the respective specific cluster which interpret it according to the definition in the clusters Scene Table Extensions section. In the cluster, the values of these attributes transition from the current value to the value in the activated scene Extension Field for that cluster. Depending on the attribute this transition may not happen instantaneously, rather may happen continuously as defined by the Transition Time in the scene definition.

The Scenes cluster requires the use of group addressing so the Groups cluster is usually on the same endpoint as the Scenes cluster. For instance, a given endpoint may have the Groups, Scenes, OnOff, and Window Covering clusters (as well as Basic). For more information see section 3.7 Scenes of [R2] and the header file zcl.scenes.h.

4.4 Identify cluster

During network commissioning it is important to be able to know each node.

The identify cluster assists in this process by introducing an identification mode that is both externally visible, for instance a flashing LED, blinking display backlight, etc. and accessible over the air. Once started, the identification mode stays active for an interval given by Identify Time after which it automatically ends. The Identify cluster can be used by itself, but it is also integral to the Touchlink procedure.

See section 3.5 Identify of [R2] and the header file `zcl.identify.h`.

4.5 Alarms cluster

The alarms are conditions that occur within certain clusters that are of interest externally to other devices on the network. Each alarm condition has an associated alarm code, defined in that cluster and often associated with an attribute or attributes. Clusters that support alarms and defined alarm codes require their endpoint also has the Alarms cluster and this is specified in the respective cluster section.

The device temperature cluster for instance, can generate two alarms: Device Temperature Too Low (code 0x00) and Device Temperature Too High (code 0x01) both associated with the device temperature. Each of these alarms is controlled by two attributes, a Threshold and a Dwell time, that are used in the determination of when an alarm condition has occurred. When the device temperature cluster determines that an alarm condition (Low or High) has occurred it posts the corresponding alarm code to the Alarms cluster on the same endpoint on which it resides. When the originating cluster makes the determination of an alarm condition (which depends on the cluster definition) it posts the alarm by calling: ZbZclClusterSendAlarm() (defined in zcl.h) inside the originating cluster.

The alarms cluster records the alarm code and originating cluster ID in a query-able alarm log. Note that alarm codes can only be interpreted in the context of the originating cluster.

After adding an entry in its alarm Log, the Alarms cluster sends an Alarm command to any clients bound to the Alarms cluster on the endpoint. In this way a client need only bind to the Alarms cluster to receive alarms from all clusters on that endpoint.

Additionally, some clusters support an alarm mask. This allows an external client to enable and disable individual alarms. Note that this alarm mask is global, enabling or disabling an alarm in the alarm mask affects any and all clients.

To summarize some clusters support alarms and define their own alarm codes for conditions that may be of interest to external clients. When a cluster supports alarms, it requires the Alarms cluster to also be present on the same endpoint to log and send the alarm. Most clusters that support alarms also have an alarm mask attribute that allows one external client to enable or suppress reporting of specific alarms. For more information see section 3.11 Alarms in [R2] and the header file zcl.alarms.h

The alarms server cluster applies a timestamp to alarms as they are added to the alarm log. This time stamp is provided by the time cluster which must be provided on the same endpoint and a pointer to the time cluster must be provided to the alarm server cluster when it is created via the `ZbZclAlarmServerAlloc()` call.

5 Special dedicated endpoint clusters

Certain special clusters reside on dedicated endpoints – endpoints which are exclusively for the purpose of that specific function. These include the Touchlink, CBKE and green power clusters. In the case of touchlink and green power one of the reasons they reside on a dedicated endpoint is because they use a special Profile ID.

The touchlink cluster and endpoint are created when `BDB_COMMISSION_MODE_TOUCHLINK` is selected. The stack creates a special touchlink endpoint(s) as set by the user in ZbStartupT (`tl_endpoint` and `bind_endpoint`).

The smart energy applications trigger the stack to create the CBKE cluster by setting the `suite`_mask in the `security.cbke` subsection of `ZbStartupT`. When one of the suites in `suite_mask` (i.e either `ZCL_KEY_SUITE_CBKE_ECMQV` or `ZCL_KEY_SUITE_CBKE_ECMQV`). When enabled, the stack creates a CBKE endpoint which by default is 240, but can be changed by the application and attaches the CBKE cluster.

The green power (green power proxy basic) cluster is automatically created by the stack (except in smart energy builds of the stack) on the reserved endpoint 242. The proxy basic has no interaction with the application.

The smart energy and non-smart energy applications use different builds of the stack. The stack build mechanism has a number of make variables that are exported as C macros of the same name. These use the prefix `CONFIG_ZB_`. For Smart Energy builds of the stack, `CONFIG_ZB_ZCL_CBKE` is defined and neither `CONFIG_ZB_GREENPOWER, CONFIG_ZB_ZCL_TL_INITIATOR`, nor `CONFIG_ZB_ZCL_TL_TARGET` are defined. The smart energy builds automatically create the CBKE cluster as described above and do not include or support the touchlink or green power proxy basic clusters.

The non-smart energy builds of the stack do not define `CONFIG_ZB_ZCL_CBKE` and thus do not include or support CBKE. They define `CONFIG_ZB_GREENPOWER` which results in the green power proxy basic cluster and endpoint being created. They also define either `CONFIG_ZB_ZCL_TL_INITIATOR` or `CONFIG_ZB_ZCL_TL_TARGET`, which results in support for either touchlink initiator or touchlink target functionality. The smart energy, touchlink initiator and touchlink target all require separate stack builds.

Revision history

Table 3. Document revision history

Date	Version	Changes
24-Jul-2020	1	Initial release
25-Aug-2020	2	Changed the document scope from ST Restricted to public

Contents

1 General information 2
1.1 Reference documents 2
1.2 Quick start 2
1.3 Checklist 2
1.4 Callbacks and application structure 3
1.5 Clusters and endpoints 3
1.6 Client server relationship 4
1.7 Endpoints and simple descriptors 5
1.8 Types of clusters 5
1.9 Application centric, support, and special clusters 5
1.10 Commands: cluster specific versus profile wide 5
1.11 Addressing and bindings 5
1.12 Find and bind 6
2 Working with cluster templates 7
2.1 The cluster pointer - ZbZclClusterT 7
2.2 Cluster naming conventions 7
2.3 Creating a new instance of a cluster 7
2.4 Remote versus local clusters 8
2.5 Remote write attribute 8
2.6 Local write attribute 8
2.7 Default PICS settings 8
2.8 Entering PICS Information Into ZTT 9
3 Application centric clusters 10
3.1 Application centric client clusters 11
3.1.1 Accessing remote server attributes from client clusters 11
3.1.2 Sending commands to remote nodes 11
3.1.3 How clients receive reports from remote server clusters 12
3.2 Application centric server clusters 13
3.2.1 Adding optional attributes to a cluster 13

3.2.2 Updating attribute values via direct write 13
3.2.3 Using cluster attribute callbacks 14
3.3 Implementing server-side command handlers 14
4 Support clusters 16
4.1 The basic cluster 16
4.2 Groups cluster 16
4.3 Scenes cluster 16
4.4 Identify cluster 17
4.5 Alarms cluster 17
5 Special dedicated endpoint clusters 18
Revision history 19
Contents 20
List of tables 22
List of figures 23

List of tables

Table 1. STM32WB Zigbee clusters 10
Table 2. Get profile command payload 12
Table 3. Document revision history 19

List of figures

Figure 1. Client server architecture 4

IMPORTANT NOTICE – PLEASE READ CAREFULLY

STMicroelectronics NV and its subsidiaries (“ST”) reserve the right to make changes, corrections, enhancements, modifications, and improvements to ST products and/or to this document at any time without notice. Purchasers should obtain the latest relevant information on ST products before placing orders. ST products are sold pursuant to ST’s terms and conditions of sale in place at the time of order acknowledgement.

Purchasers are solely responsible for the choice, selection, and use of ST products and ST assumes no liability for application assistance or the design of Purchasers’ products.

No license, express or implied, to any intellectual property right is granted by ST herein.

Resale of ST products with provisions different from the information set forth herein shall void any warranty granted by ST for such product.

ST and the ST logo are trademarks of ST. For additional information about ST trademarks, please refer to www.st.com/trademarks. All other product or service names are the property of their respective owners.

Information in this document supersedes and replaces information previously supplied in any prior versions of this document.

© 2020 STMicroelectronics – All rights reserved