Application note

Creating manufacture specific clusters on STM32WB Series

Introduction

The purpose of this application note is to guide the end-user in the implementation of specific ZCL (Zigbee® clusters library) manufacture on STM32WB Series.

The Exegin ZSDK (Zigbee software design kit) includes templates for most existing clusters and provides a wide range of functionalities. However, some applications require the development of custom-cluster templates. This document describes the process of developing these custom-cluster templates. It describes building new ZCL clusters in the same way the Exegin ZSDK clusters are built.

It is assumed that the end-user is familiar with general Zigbee® networking [R1], the Exegin ZSDK stack reference [R3], and Using Exegin ZCL Cluster Templates [R5] (see Table 1. Reference documents).

1 General information

This document applies to STM32WB Series Arm®-based devices.

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

2 Reference documents

Below resources are public and available either on STMicroelectronics or on third parties websites.

Table 1. **Reference documents**

Reference number	Document title	Document number
[R1]	Zigbee PRO Specification Revision 22: www.zigbeealliance.org/(1)	05-2374-22
[R2]	Zigbee Cluster Library Revision 7: www.zigbeealliance.org/ (1)	07-5123-07
[R3]	ZSDK API implementation for ZigBee® on STM32WB Series(2)	AN5500
[R4]	Zigbee Smart Energy Standard Version 1.4: www.zigbeealliance.org/ (1)	07-5356-21
[R5]	How to use ZigBee® clusters templates on STM32WB Series(2)	AN5498

1. *This URL belongs to a third party. It is active at document publication, however STMicroelectronics shall not be liable for any change, move or inactivation of the URL or the referenced material.*
2. *Available at www.st.com. Contact STMicroelectronics when more information is needed.*

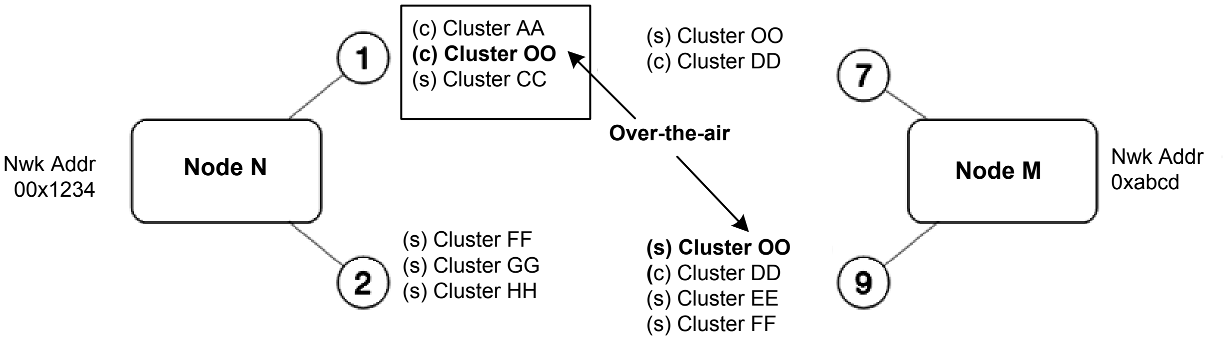
3 ZCL cluster architecture

Before covering the mechanics to implement new ZCL clusters, it is useful to review ZCL fundamentals and provide an example that can be referenced throughout this document.

The Zigbee cluster library (ZCL) defines mechanisms for applications to interact across the network and over-the-air between nodes. The functionality for a specific purpose is organized into a set called a “cluster” which defines a set of related attributes and commands. For instance, the On/Off cluster defines the functionality of devices that can be turned On or Off.

The figure below shows two nodes of a network. Nodes N and M both support cluster OO (for example: On/Off) on endpoints 1 and 9 respectively. The functionality is split between a client and server side. In the On/Off example the client can be a switch and the server can be a light. With the switch on endpoint 1 Node N controlling the light on endpoint 9 of Node M.

Figure 1. Two nodes network example



ZCL is built on APS messaging which in turn uses NWK layer functionality. For more information refer to [R3]. ZSDK includes common services on which ZCL clusters can be built. The ZCL cluster templates [R5] are built on these common services. The figure below shows the ZCL structure in relation to the application and the ZSDK stack.

Figure 2. ZCL application structure

The ZCL cluster specific functionality, for instance as defined in the cluster definition in ZCL [R2] or ZSE [R4] specifications, is implemented in the cluster template as shown in yellow in Figure 2. ZCL application structure.

These templates are completed by adding the device specific details in the client or server cluster application, as shown in green in Figure 2. ZCL application structure. For example, the details of accessing the physical hardware switch on endpoint 1 of node N or the physical light on endpoint 9 in node M of Figure 1. Two nodes network example.

4 Attribute end-to-end sequence of operations

An example of client application as shown below reads an attribute from the remote server by calling the ZCL cluster base function `ZbZclReadReq()`. This provides the client cluster, read request structure, and a callback function to be invoked with the result shown as follows.

```
static void
read_onoff_callback(const ZbZclReadRspT *rsp, void *arg)
{
    struct application *app = (application *)arg;

    if(rsp->status == ZCL_STATUS_SUCCESS) {
        if(rsp->attr[0].status == ZCL_STATUS_SUCCESS) {
            app->remote_onoff_state = pletoh16(rsp->attr[0].value[0]);
        }
        else { /* handle error */
        ...
    }
    else { /* handle error */
    ...
}

.
.
.

    enum ZclStatusCodeT status;
    ZbZclReadReqT req;

    memset(&req, 0, sizeof(req));
    req.dst = ZbApsAddrBinding;
    req.attr[0] = ZCL_ONOFF_ATTR_ON_TIME;
req.count = 1;
status = ZbZclReadReq(app->client_cluster, &req, read_onoff_callback,    &application)
```

Calling `ZbZclReadReq()` sends the ZCL attribute request over-the-air to the server. This example uses binding and assumes that node N contains a binding on the OnOff cluster from its local endpoint 1 to Node M endpoint 9. If the server cluster supports the requested attribute, the value for the attribute is returned in a ZCL attribute read response command. If the attribute does not exist, a ZCL default response is returned with`UNSUPPORTED_ATTRIBUTE` status (0x86). Attributes are added to a cluster either during initialization (mandatory attributes), or using `ZbZclAttrAppendList()`. On reception of the ZCL read attributes response, the callback is called.

The structure to define an attribute is with `struct ZbZclAttrT`. Attributes are initialized as a list of these structures and provided to `ZbZclAttrAppendList()` to be attached to a cluster. An example of server cluster application or server template is shown below.

```
static const struct ZbZclAttrT attr_list[] = {
    ...
    {
        ZCL_ONOFF_ATTR_ON_TIME, ZCL_DATATYPE_UNSIGNED_16BIT,
        ZCL_ATTR_FLAG_NONE, 0, NULL, {0, 0}, {0, 0}
},
...
```

This is added to the cluster using:

```
ZbZclAttrAppendList(cluster, attr_list, ZCL_ATTR_LIST_LEN(attr_list))
```

Note: *In some cases, client clusters have their own separate attributes. The foregoing discussion stands, unaltered with the names client and server reversed.*

The above example of server cluster application or server template shows the attribute callback being set to NULL. In this case the ZCL cluster base responds to the read request with whatever value that attribute has at the moment the request is received. The server application updates the local value through a ZCL cluster base call such as `ZbZclAttrIntegerWrite()`. The value that the server application has previously set is the value that is used in the response to the read request. The server application is decoupled from the client access by the cluster base.

Another approach is for the server to provide an attribute callback as shown below in the server cluster application example.

```
/* hardware register, little endian, two bytes*/
extern volatile uint8_t *on_time;

enum ZclStatusCodeT on_time_cb(struct ZbZclClusterT *clusterPtr, struct ZbZclAttrCbInfoT
*info)
{
    uint8_t *data = info->attr_data;

    switch(info->type) {
        case ZCL_ATTR_CB_TYPE_READ:
            /* read from hardware */
            data[0] = on_time[0];
            data[1] = on_time[1]
            break;

        case ZCL_ATTR_CB_TYPE_WRITE:
            /* write to hardware */
            on_time[0] = data[0];
            on_time[1] = data[1];
            break;

        case ZCL_ATTR_CB_TYPE_NOTIFY:
            on_time[0] = data[0];
            on_time[1] = data[1];
            break;
    }

    return ZCL_STATUS_SUCCESS;
}


static const struct ZbZclAttrT attr_list[] = {
    ...
    {
        ZCL_ONOFF_ATTR_ON_TIME, ZCL_DATATYPE_UNSIGNED_16BIT,
        ZCL_ATTR_FLAG_NONE, 0, on_time_cb, {0, 0}, {0, 0}
    },
    ...
```

When providing a callback, the server application is called every time the attribute is read or written. This is enabled by setting the flags `ZCL_ATTR_FLAG_CB_READ` and `ZCL_ATTR_FLAG_CB_WRITE` in the attribute definition above. It is recommended that either the application set both the read and write flags and provide a callback to handle both read and write as shown, or that the callback is set to NULL and these flags not to be set. With no callback the value is managed completely in the stack. With a callback and both flags set the value is managed completely in the application.

Although possible, it not recommended to handle just read or just write requests in the application because without careful attention it is possible read and written values are different.

Once the server call completes, the cluster base prepares and sends a response message over-the-air back to the client. The status and value in the case of a read and the status for a write. The client stack receives the response message. The cluster base on the client processes it and if there is a callback provided (such as `read_onoff_callback` in the previous example) it is invoked with the status, and for a "read the value".

5 Command end-to-end walkthrough

When a client sends a command request to the server, the process is similar to the detailed one in Section 4 Attribute end-to-end sequence of operations. However, rather than sending a command using a ZCL base function, the client template provides a command specific request function capable of ZCL payload for the command.

An example of this walkthrough of the client sending a `GetCalendar` command request to the server is shown in Table 2, Table 3 and Table 4 . If this is successful it produces a PublishCalendar response back to the client. Refer to [R4] for details on these Calendar cluster commands.

5.1 Client-side command generation

Section D.9.2.4.1.1 [R4] defines the `GetCalendar` command as shown in the table below.

Table 2. GetCalendar command

Octets	Data type	Field name
4	UTC time	Earliest start time (M)
4	Unsigned 32-bit integer	Min. issuer event ID (M)
1	Unsigned 8-bit integer	Number of calendars (M)
1	8-bit enumeration	Calendar type (M)
4	Unsigned 32-bit integer	Provider ID (M)

The `PublishCalendar` response from the server defined in D.9.2.3.1.1 [R4] is as follows.

Table 3. PublishCalendar response - part one

Octets	Data type	Field name
4	Unsigned 32-bit integer	Provider ID (M)
4	Unsigned 32-bit integer	Issuer event ID (M)
4	Unsigned 32-bit integer	Issuer calendar ID (M)
4	UTC time	Start time (M)
1	8-bit enumeration	Calendar type (M)

Table 4. PublishCalendar response - part two

Octets	Data type	Field name
1	Unsigned 8-bit integer	Calendar time reference (M)
1..13	Octet string	Calendar name (M)
1	Unsigned 8-bit integer	Number of seasons (M)
1	Unsigned 8-bit integer	Number of week profiles (M)
1	Unsigned 8-bit integer	Number of day profiles (M)

When the client application decides to request a calendar, it sends a `Get Calendar`. This is shown below in the cluster client application example.

```
static void get_cal_cb (struct ZbZclCommandRspT *rsp, void *arg)
{
    struct application app = (struct application *)arg;


}

...

    enum ZclStatusCodeT status;
    struct ZbZclCalClientGetCalendarT req;

    memset(&req, 0, sizeof(req));
    req.earliestStartTime = app->calendar_start;
    req.minIssuerEventId = ZCL_INVALID_UNSIGNED_32BIT; /* all ids */
    req.numCalendars = 1;
    req.calendarType = 0x02; /* delivered and received */
    req.providerId = app->provider_id;

status = ZbZclCalClientCommandGetCalReq(cluster, ZbApsAddrBinding, &req, get_cal_cb, app);
```

If the parameters to `ZbZclCalClientCommandGetCalReq()` are valid, the status returned is `ZB_STATUS_SUCCESS` and an attempt is made to send the `GetCalendar` request as shown in Table 2. GetCalendar command. Otherwise an error status is returned and the request is not sent.

When the return is `ZB_STATUS_SUCCESS` the callback (`get_cal_cb` in the example above) is called to indicate the result of the client's attempt, and if successful, the server's response in the callback.

The callback required to take into account the behavior of the server as defined in the specification. For some ZCL request commands the required response is a detailed response, such as the `PublishCalendar` response to the `GetCalendar` request above. In other cases, the ZCL response is primarily a status, sometimes with additional information. When no ZCL message response is defined, the ZCL specicification requires the server to return the generic default response message.

Moreover, because the response is over-the-air, it is possible the client request is:

- not received by the server,
- the server can reject or unable to process the request,
- or the response does not reach the client

The callback can detect each of these conditions and take the appropriate action as required. The action of the application is dependent on each of these possibilities and on the specific requirement of the application.

The client can:

- fire-and-forget
- retry
- or take more drastic actions such as rejoining or initiating a frequency agility procedure as appropriate.

The choice depends entirely on the requirements of the application.

Using the previous example, the following shows the possible client side outcomes of sending the `ZbZclCalClientCommandGetCalReq()` in the callback.

1. When the request was successful and responded to with a ZCL message over-the-air by the server, the value of `rsp.status` is`ZCL_STATUS_SUCCESS` . Additionally:
    1. The frame type is “cluster” that is `(rsp.hdr.frameCtrl.frameType & ZCL_FRAMECTRL_TYPE) == ZCL_FRAMETYPE_CLUSTER`
    2. The ZCL command ID is the expected command ID of the response. In this case a Publish Calendar is expected:`rsp.hdr.cmdId == ZCL_CAL_SVR_PUBLISH_CALENDAR`
    3. After performing these checks, the application can parse the response using the helper function provided. In this case to parse the Publish Calendar response the application can call `ZbZclCalClientParsePublishCalendar()` with the payload `rsp->payload` and `rsp->length`

Note: Some ZCL command responses contain embedded status codes which must also be checked.

2. When the request was successful, and there is no defined ZCL message response from the server, or there was an error with the command, the server then responds with a Default Response message. As with the previous case the value of `rsp.status` is `ZCL_STATUS_SUCCESS`. Additionally:
    1. The frame type instead is`ZCL_FRAMETYPE_PROFILE`
    2. And the command ID is be `ZCL_COMMAND_DEFAULT_RESPONSE`
    3. `ZbZclParseDefaultResponse()` can be used to parse the default response
    4. A default response with a status other than `ZCL_STATUS_SUCCESS` denotes a failed command response from the server.

Note: The Disable Default Response flag of frame control of a ZCL message is often set to 1 causing a “success” Default Response only when there is no other ZCL Response. However, if set to 0 a Default Response is always sent even when the response is a ZCL message.

3. If there is an error in the transmission of the request or if the server does not respond, the value of `rsp.status` is `ZCL_STATUS_FAILURE`. In this case the value of `rsp.aps_status` must be checked and contains the APS or NWK layer status code. Some examples include:
    1. The `aps_status` is `ZB_APS_STATUS_NO_ACK`. This means that the client successfully sent the request and it was received at the network level, but the server did not acknowledge receipt at the APS layer. When the server can interpret but rejects the request it normally sends a Default Response with an error status as above. When the server cannot interpret the message, for example:
        - the destination endpoint does not exist on the server
        - or the message does not meet the minimum security/encryption requirements
        - the server does not or cannot acknowledge receipt
        - an APS layer acknowledgement is not sent.

       Then a timeout occurs and an `aps_status` of `ZB_APS_STATUS_NO_ACK` is returned to the application.
    2. If when sending the request, the client is unable to even reach the next hop in the route, the `aps_status` becomes `ZB_WPAN_STATUS_NO_ACK`. There are several layers of automatic retry built-in. In the unlikely event that this is the `aps_status`, the cause is most likely that the client has lost connectivity with the network completely.

5.2 Server-side command reception and processing

When the client sends the GetCalendar request over-the-air and it is received by the server, the stack receives the message see Node M in Section 3 , passing through the NWK and APS layers to the ZCL cluster base. When the application created this instance of the cluster, code in the cluster template registered a template level command handler and a set of cluster server application callbacks, one for each command.

```
static enum ZclStatusCodeT
get_calendar(struct ZbZclClusterT *clusterPtr, void *arg,
    struct ZbZclCalClientGetCalendarT *req,
    struct ZbZclAddrInfoT *srcInfo)
{
    struct ZbZclCalServerPublishCalendarT rsp;

    memset(&rsp, 0, sizeof(rsp));
    /* fill in response */
    ZbZclCalServerSendPublishCalendar(clusterPtr, srcInfo, &rsp);
    return ZCL_STATUS_SUCCESS_NO_DEFAULT_RESPONSE;
}

…
callbacks.get_calendar = get_calendar;
…
cluster = ZbZclCalServerAlloc(zb, endpoint, &callbacks, arg)
…
```

```
struct ZbZclClusterT *
ZbZclCalServerAlloc(struct ZigBeeT *zb, uint8_t endpoint, struct ZbZclCalServerCallbacksT
*callbacks, void *arg)
{
    ...
    clusterPtr->cluster.command = zcl_calendar_server_command;

    ...

    ZbZclCalServerConfigCallbacks(&clusterPtr->cluster, callbacks);
```

In the above cluster template example the ZCL cluster base invokes the `zcl_calendar_server_command()` callback, which contains code to unpack every supported command. For unsupported commands the callback must return `ZCL_STATUS_UNSUPP_MFR_CLUSTER_COMMAND` (or `ZCL_STATUS_UNSUPP_CLUSTER_COMMAND` for clusters defined in the ZCL specification [R2]). These return codes trigger the ZCL cluster base to send the corresponding default response back to the client.

Otherwise, when the command is supported, the cluster template command handler unpacks the ZCL command payload into a struct `ZbZclCalClientGetCalendarT` data structure and calls the provided cluster server application`get_calendar()callback`.

For the GetCalendar command, the`get_calendar()` callback is responsible for determining which calendars match the criteria in the request, `ZbZclCalClientGetCalendarT *req`, and returning the matching calendars by calling `ZbZclCalServerSendPublishCalendar()`. Implementing this forms the ZCL PublishCalendar message and returns it to the client. This server callback then exits with `ZCL_STATUS_SUCCESS_NO_DEFAULT_RESPONSE` ending the server-side transaction and informing the ZCL cluster base that it has already sent a response (ZCL PublishCalendar) and “success” Default Response must only be sent if required.

If the cluster server application callback is successful, but a ZCL Message response is not sent then the callback should end with `ZCL_STATUS_SUCCESS`, generating a “success” Default Response. Otherwise, a non-success `ZCL_STATUS` must be returned, which always generates a Default Response with the corresponding status code. However, when implementing the ZCL specification, only the status codes detailed in the specification must be returned.

Revision history

Table 5. **Document revision history**

Date	Version	Changes
17-Jul-2020	1	Initial release

Contents

List of tables

List of figures

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