Z-Stack Home
Developer's Guide

Document Number: SWRA441

Texas Instruments, Inc.
San Diego, California USA

Version	Description	Date
1.0	Initial release	11/21/2013
1.1	Cleanup for Z-Stack Home 1.2.1	07/02/2014

TABLE OF CONTENTS

LIST OF FIGURES

1. Introduction

1.1 Purpose

This document explains the main components of the Texas Instruments Z-Stack Home release and its functionality. It explains the configurable parameters in Z-Stack Home and how they may be changed by the application developer to suit particular application requirements.

1.2 Scope

This document enumerates the parameters required to be modified in order to create a ZigBee HA compatible product, and the various ZigBee HA related parameters which may be modified by the developer. This document does not provide details of other profiles and functional domains, nor the underlying Z-Stack infrastructure. Furthermore, this document doesn't explain the ZigBee HA concepts.

1.3 Definitions, Abbreviations and Acronyms

Term	Definition
API	Application Programming Interface
MT	Z-Stack's Monitor and Test Layer
OSAL	Z-Stack's Operating System Abstraction Layer
OTA	Over-the-air
ZCL	ZigBee Cluster Library
ZHA	ZigBee Home Automation
ZC	ZigBee Coordinator
ZR	ZigBee Router
ZED	ZigBee End Device

1.4 Applicable Documents

[1] ZigBee document 05-3520-29 ZigBee Home Automation Specification.

[2] ZigBee document 07-5340-13, ZigBee Home Automation Test Specification.

[3] Texas Instruments document SWRU354, Z-Stack Home Sample Application User's Guide.

[4] Texas Instruments document SWRA176, Z-Stack Developer's Guide.

[5] Texas Instruments document SWRA194, OSAL API.

[6] Texas Instruments document SWRA353, Z-Stack OTA Upgrade User's Guide.

2. Overview

2.1 Introduction

This document refers to Z-Stack™ Home Sample Applications. Each one of the Z-Stack Sample Applications is a simple head-start to using the TI distribution of the ZigBee Stack in order to implement a specific Application Object.

This document is intended as a generic overview. For a hands-on user's guide, please refer to *Z-Stack Home Sample Application User's Guide* [3].

Each sample application uses the minimal subset of ZDO Public Interfaces that it would take to make a Device reasonably viable in a ZigBee network. In addition, all sample applications utilize the essential OSAL API functionality: inter and intra-task communication, by sending and receiving messages, setting and receiving task events, setting and receiving timer callbacks, using dynamic memory, as well as others. In addition, every sample application makes use of the HAL API functionality, e.g. for controlling LED's. Thus, any sample application serves as a fertile example from which to copy-and-paste, or as the base code of the user's application to which to add additional Application Objects.

Any Application Object must sit overtop of a unique Endpoint; and any Endpoint is defined by a Simple Descriptor. The numbers used for the Endpoints in the Simple Descriptors in the sample applications have been chosen arbitrarily.

Each sample application instantiates only one Application Object, and therefore, only supports the one corresponding Profile. But keep in mind that two or more Application Objects may be instantiated in the same Device. When instantiating more than one Application Object in the same Device, each Application Object must implement a unique Profile Id and sit overtop of a unique Endpoint number. The sample applications meet the unique Id's and Endpoint numbers requirement and could be combined into one Device with minor modifications.

The following paragraphs describe the OSAL Tasks:

2.1.1 Initialization

OSAL is designed and distributed as source so that the entire OSAL functionality may be modified by the Z-Stack user. The goal of the design is that it should not be necessary to modify OSAL in order to use the Z-Stack as distributed. The one exception is that the OSAL Task initialization function, osalInitTasks(), must be implemented by the user. The sample applications have implemented this function in a dedicated file, named according to the following pattern: OSAL_"Application Name".c (e.g. OSAL_SampleLight.c). The BSP will invoke osalInitTasks() as part of the board power-up and Z-Stack initialization process.

2.1.2 Organization

As described in the Z-Stack OSAL API [5] document, the OSAL implements a cooperative, round-robin task servicing loop. Each major sub-system of the Z-Stack runs as an OSAL Task. The user must create at least one OSAL Task in which their application will run. This is accomplished by adding their task to the task array [tasksArr defined in OSAL_"Application Name".c] and calling their application's task initialization function in osalInitTask(). The sample applications clearly show how the user adds a task to the OSAL system.

2.1.3 Task Priority

The tasks are executed in their order of placement in the task array [tasksArr in OSAL_"Application Name".c]. The first task in the array has the highest priority.

2.1.4 System Services

The OSAL and HAL system services are keyboard (switch press) notification and serial port activity notification. The system services are exclusive, meaning that each of them may be registered by no more than a single OSAL Task. Different system services may be registered by the same OSAL Task, or by different OSAL Tasks. By default, none of the Z-Stack Tasks register for any of the system services – they are both left for the user's application.

2.1.5 Application Design

The user may create one task for each Application Object instantiated or service all of the Application Objects with just one task. The following are some of the considerations when making the aforementioned design choice.

2.1.5.1 One OSAL Task per many Application Objects

These are some of the pros & cons of the one-per-many design:

- Pro: The action taken when receiving an exclusive task event (switch press or serial port) is simplified.
- Pro: The heap space required for many OSAL Task structures is saved.
- Con: The action taken when receiving an incoming AF message or an AF data confirmation is complicated – the burden for de-multiplexing the intended recipient Application Object is on the single user task.
- Con: The process of service discovery by Match Descriptor Request (i.e. auto-match) is more complicated – a static local flag must be maintained in order to act properly on the ZDO_NEW_DSTADDR message.

2.1.5.2 One OSAL Task per one Application Object

These pros & cons of the one-per-one design are the inverse of those above for the one-per-many design:

- Pro: An incoming AF message or an AF data confirmation has already been de-multiplexed by the lower layers of the stack, so the receiving Application Object is the intended recipient.
- Con: The heap space required for many OSAL Task structures is incurred.
- Con: The action taken when receiving an exclusive task event may be more complicated if two or more Application Objects use the same exclusive resource.

2.1.6 Mandatory Methods

Any OSAL Task must implement two methods: one to perform task initialization and the other to handle task events.

2.1.6.1 Task Initialization

In the sample applications, the callback function to perform task initialization is named according to the following pattern: zcl"Application Name"_Init (e.g. `zclSampleLight_Init()`). The task initialization function should accomplish the following:

- Initialization of variables local to or specific for the corresponding Application Object(s). Any long-lived heap memory allocation should be made here in order to facilitate more efficient heap memory management by the OSAL.
- Instantiation of the corresponding Application Object(s) by registering with the AF layer (e.g. afRegister()).
- Registration with the applicable OSAL or HAL system services (e.g. RegisterForKeys()).

2.1.6.2 Task Event Handler

In the sample applications, the callback function to perform task event handling is named according to the following pattern: zcl"Application Name"_event_loop (e.g. `zclSampleLight_event_loop()`). Any OSAL Task can define up to 15 events in addition to the mandatory event.

2.1.7 Mandatory Events

One task event, `SYS_EVENT_MSG` (0x8000), is reserved and required by the OSAL Task design.

2.1.7.1 SYS_EVENT_MSG (0x8000)

The global system messages sent via the SYS_EVENT_MSG are specified in ZComDef.h. The task event handler should process the following minimal subset of these global system messages. The recommended processing of the following messages should be learned directly from the sample application code or from the study of the program flow in the sample application.

2.1.7.1.1 AF_DATA_CONFIRM_CMD

This is an indication of the over-the-air result for each data request that is successfully initiated by invoking AF_DataRequest(). ZSuccess confirms that the data request was successfully transmitted over-the-air. If the data request was made with the AF_ACK_REQUEST flag set, then the ZSuccess confirms that the message was successfully received at the final destination. Otherwise, the ZSuccess only confirms that the message was successfully transmitted to the next hop.[i]

2.1.7.1.2 AF_INCOMING_MSG_CMD

This is an indication of an incoming AF message.

2.1.7.1.3 KEY_CHANGE

This is an indication of a key press action.

2.1.7.1.4 ZDO_STATE_CHANGE

This is an indication that the network state has changed.

2.1.7.1.5 ZDO_CB_MSG

This is message is sent to the sample application for every registered ZDO response message [ZDO_RegisterForZDOMsg()].

2.2 Network Formation

A sample application compiled as a Coordinator will form a network on one of the channels specified by the DEFAULT_CHANLIST. The Coordinator will establish a random Pan ID based on its own IEEE address or on ZDAPP_CONFIG_PAN_ID if it is not defined as 0xFFFF. A sample application compiled as a Router or End-Device will try to join a network on one of the channels specified by DEFAULT_CHANLIST. If ZDAPP_CONFIG_PAN_ID is not defined as 0xFFFF, the Router will be constrained to join only the Pan ID thusly defined. Note the unexpected result achieved because of the difference in behavior between a Coordinator and a Router or End-Device when ZDAPP_CONFIG_PAN_ID is not defined as 0xFFFF. If ZDAPP_CONFIG_PAN_ID is defined as a valid value less than or equal to 0xFFFE, then the Coordinator will only attempt to establish a network with the specified Pan Id. Thus, if the Coordinator is constrained to one channel, and the specified Pan Id has already been established on that channel, the newly starting Coordinator will make successive changes until it achieves a unique Pan Id. A Router or End-Device newly joining will have no way of knowing the value of the "de-conflicted" Pan Id established, and will therefore join only the Pan Id specified. A similarly challenging scenario arises when the permitted channel mask allows more than one channel and the Coordinator cannot use the first channel because of a Pan Id conflict – a Router or End-Device will join the specified Pan Id on the first channel scanned, if allowed.

2.2.1 Auto Start

A device will automatically start trying to form or join a network as part of the BSP power-up sequence. If the device should wait on a timer or other external event before joining, then HOLD_AUTO_START must be defined. In order to manually start the join process at a later time, invoke ZDOInitDevice().

2.2.2 Network Restore

Devices that have successfully joined a network can "restore the network" (instead of reforming by over-the-air messages) even after losing power or battery. This automatic restoration can be enabled by defining NV_RESTORE and/or NV_INIT.

2.2.3 Join Notification

The device is notified of the status of the network formation or join (or any change in network state) with the `ZDO_STATE_CHANGE` message mentioned above.

2.3 Common Application Framework / Program Flow

This section describes the initialization and main task processing concepts that all sample applications share. This section should be read before moving on to the sample applications. For code examples in this section, we will use "SampleLightApp". provided in the Z-Stack Home installer.

2.3.1 Initialization

During system power-up and initialization, the task's initialization function will be invoked (see 2.1.6.1). The structure of this function is explained below, together with some code examples:

```
zclSampleLight_TaskID = task_id;
```

The task ID is assigned by the OSAL and is given to the task via the parameter of the task's init function. The application must remember this task id, as it will use it later for self-triggering using OSAL timers, events and messages (e.g. when the task sends a message to itself). Such self-triggering is usually used for dividing long processes into smaller chunks that are then invoked with some timed delay between them. This "cooperative" behavior allows other tasks to share the CPU time and prevents "starvation".

```
zclHA_Init( & zclSampleLight_SimpleDesc );
```

The SampleLightApp Application Object is instantiated by the above code. This allows the AF layer to know how to route incoming packets destined for the profile/endpoint – it will do so by sending an OSAL SYS_EVENT_MSG-message (AF_INCOMING_MSG_CMD) to the task (task ID).

```
RegisterForKeys( zclSampleLight_TaskID );
```

The sample application registers for the exclusive system service of key press notification.[ii]

2.3.2 Event Processing

Whenever an OSAL event occurs for the sample application, the event processing functions (see 2.1.6.2), will be invoked in turn from the OSAL task processing loop. The parameter to the task event handler is a 16-bit bitmask; One or more bits may be set in any invocation of the function. If more than one event is set, it is strongly recommended that a task should only act on one of the events (probably the most time-critical one, and almost always, the SYS_EVENT_MSG as the highest priority one). The unhandled event will cause a new invocation of this task event handler after all the other handlers get the chance to process their events.

```
if ( events & SYS_EVENT_MSG )
{
 MSGpkt = (afIncomingMSGPacket_t*)osal_msg_receive(zclSampleLight_TaskID );
 while ( MSGpkt )
 {
  …
```

Notice that although it is recommended that a task only act on one of possibly many pending events on any single invocation of the task processing function, it is also recommended (and implemented in the sample applications) to process all of the possibly many pending SYS_EVENT_MSG messages all in the same "time slice" from the OSAL.

```
        switch ( MSGpkt->hdr.event )
```


It is recommended that a task implement a minimum subset of the possible types of SYS_EVENT_MSG messages.[iii] This recommended subset is described below.

```
case KEY_CHANGE:
  zclSampleLight_HandleKeys ( ((keyChange_t *)MSGpkt)->state,
                              ((keyChange_t *)MSGpkt)->keys );
  break;
```

If an OSAL Task has registered for the key press notification, any key press will be received as a KEY_CHANGE system event message. There are two possible paths of program flow that would result in a task receiving this KEY_CHANGE message.

The program flow that results from the physical key press is the following:

- HAL detects the key press state (either by an H/W interrupt or by H/W polling.)
- The HAL OSAL task detects a key state change and invokes the OSAL key change callback function.
- The OSAL key change callback function sends an OSAL system event message (KEY_CHANGE) to the Task Id that registered to receive key change event notification (RegisterForKeys().)

```
case AF_DATA_CONFIRM_CMD:
  // The status is of ZStatus_t type [defined in ZComDef.h]
  // The message fields are defined in AF.h
  afDataConfirm = (afDataConfirm_t *)MSGpkt;
  sentEP = afDataConfirm->endpoint;
  sentStatus = afDataConfirm->hdr.status;
  sentTransID = afDataConfirm->transID;
```

Any invocation of AF_DataRequest() that returns ZSuccess will result in a "callback" by way of the AF_DATA_CONFIRM_CMD system event message.

The sent Transaction Id (sentTransID) is one way to identify the message. Although the sample application will only use a single Transaction Id counter, it might be useful to keep a separate Transaction Id counter for each different endpoint or even for each cluster ID within an endpoint for the sake of message confirmation, retry, disassembling and reassembling, etc. Note that any transaction ID state variable (counter) that is passed to AF_DataRequest() gets incremented by this function upon success (thus it is a parameter passed by reference, not by value.)

The return value of AF_DataRequest() of ZSuccess signifies that the message has been accepted by the Network Layer which will attempt to send it to the MAC layer which will attempt to send it over-the-air.

The sent Status (sentStatus) is the over-the-air result of the message. ZSuccess signifies that the message has been delivered to the next-hop ZigBee device in the network. If AF_DataRequest() was invoked with the AF_ACK_REQUEST flag, then ZSuccess signifies that the message was delivered to the destination address. Unless the addressing mode of the message was indirect (i.e. the message was sent to the network reflector to do a binding table lookup and resend the message to the matching device(s)), in which case ZSuccess signifies that the message was delivered to the network reflector. There are several possible sent status values to indicate failure.[iv]

```
case ZDO_STATE_CHANGE:
  zclSampleLight_NwkState = (devStates_t)(MSGpkt->hdr.status);
  if (  (zclSampleLight_NwkState == DEV_ZB_COORD) ||
        (zclSampleLight_NwkState == DEV_ROUTER)   ||
        (zclSampleLight_NwkState == DEV_END_DEVICE) )
  {
     giLightScreenMode = LIGHT_MAINMODE;
    ...
  }
  break;
```

Whenever the network state changes, all tasks are notified with the system event message ZDO_STATE_CHANGE.


```
        // Release the memory
        osal_msg_deallocate( (uint8 *)MSGpkt );
```

Notice that the design of the OSAL messaging system requires that the receiving task re-cycle the dynamic memory allocated for the message.

The task's event processing function should try to get the next pending SYS_EVENT_MSG message:

```
    // Get the next message
    MSGpkt = (afIncomingMSGPacket_t*)osal_msg_receive(zclSampleLight_TaskID);
  }
```

After processing an event, the processing function should return the unprocessed events, e.g. after processing SYS_EVENT_MSG, the following should be returned:

```
      // Return unprocessed events
      return ( events ^ SYS_EVENT_MSG);

    }
```

3. The Home Automation Profile and the Sample Applications

3.1 Introduction

The HA Profile, defined as a standard ZigBee profile with Profile ID 0x0104, relies upon the ZCL in general and specifically on the Foundation and General function domain.

This section describes the implementation of the Sample Applications, for a detailed description on how to use them, refer to [3].

3.2 Initialization

As described above, the user must add at least one task (to the OSAL Task System) to service the one or many Application Objects instantiated. But when implementing an HA Application Object, a task dedicated to the ZCL must also be added (see the "OSAL Tasks" section) in the order shown in the HA Sample Applications (i.e. before any other tasks using ZCL).

In addition to the user task initialization procedure studied in the generic example in the previous section, the initialization of an HA Application Object also requires steps to initialize the ZCL. The following HA Sample Applications (Light and Switch) implement this additional initialization in the task initialization function (e.g. `zclSampleLight_Init()`) as follows.

- Register the *simple descriptor* (e.g. zclSampleLight_SimpleDesc) with the HA Profile using `zclHA_Init()`.
- Register the *command callbacks table* (e.g. `zclSampleLight_CmdCallbacks)` with the General functional domain using `zclGeneral_RegisterCmdCallbacks()`.
- Register the *attribute list* (e.g. `zclSampleLight_Attrs)` with the ZCL Foundation layer using `zcl_registerAttrList()`.

3.3 Software Architecture

Applications send data through various send functions (*e.g. zclGeneral_SendOnOff_CmdToggle*). Applications receive data in callback functions after they register the callbacks. Each cluster has its own register and set of callback functions (*e.g. zclSampleLight_OnOffCB*).

The way Z-Stack communicates with other devices can be described by **Figure 1**. Clients initiate a command or request a response. Servers act on a command or deliver the response.

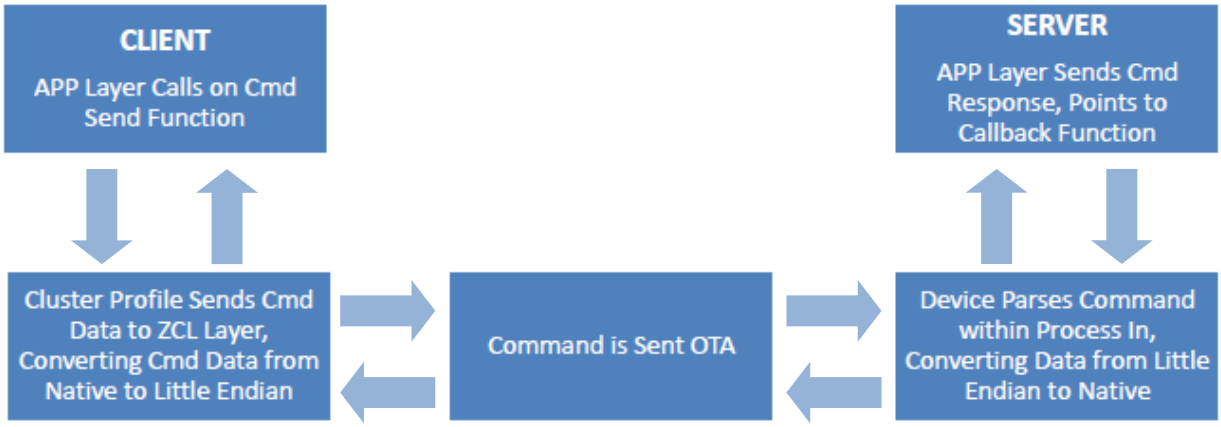

Figure 1 - Command Flowchart

A cluster's callback functions are registered within the application's initialization function (*e.g. zclSampleLight_Init*) by including each application endpoint and a pointer to each callback function and calling on a register function (*e.g. zclGeneral_RegisterCmdCallbacks*). Each cluster, or set of clusters, has its own register command. Only those callbacks defined as non-NULL will receive the data indication or response from Z-Stack.

As an example of a callback function, if a client sends a BasicReset command to a server, the application's registered BasicReset callback function on the server side is called (*e.g. zclSampleDoorLockController_BasicResetCB),* which can in turn reset the device to factory defaults.

The callback function in an application provides additional processing of a command that is specific to that application.

Under the hood, data sent via a *Send* function is converted from native format to little endian over-the-air format. Data received over-the-air is converted from over-the-air to native format on the *ProcessIn* functions. From an application viewpoint, all data is in native structure form.

For the *Send* and *ProcessIn* functions to be available, clusters, or groups of clusters must be enabled, either in the f8wZCL.cfg file, or in the compiler's predefined constants section.

3.4 File Structure

The following directories hold most of the Home Automation related code:

3.4.1 On Off Light Sample Application (Light and Switch).

1. IAR workspace files for Light project:

   \Projects\zstack\HomeAutomation\SampleLight\<platform>

2. Source files for Light project:

   \Projects\zstack\HomeAutomation\SampleLight\Source

3. IAR workspace files for Switch project:

   \Projects\zstack\HomeAutomation\SampleSwitch\<platform>

4. Source files for Switch project:

   \Projects\zstack\HomeAutomation\SampleSwitch\Source

3.4.2 DoorLock Sample Application (DoorLock and DoorLockController).

1. IAR workspace files for DoorLock project:

   \Projects\zstack\HomeAutomation\SampleDoorLock\<platform>

2. Source files for DoorLock project:

   \Projects\zstack\HomeAutomation\SampleDoorLock\Source

3. IAR workspace files for DoorLockController project:

   \Projects\zstack\HomeAutomation\SampleDoorLockController\<platform>

4. Source files for DoorLockController project:

   \Projects\zstack\HomeAutomation\SampleDoorLockController\Source

3.4.3 HVAC Sample Application (Thermostat, TemperatureSensor and HeatingCoolingUnit).

1. IAR workspace files for Thermostat project:

   \Projects\zstack\HomeAutomation\SampleThermostat\<platform>

2. Source files for Thermostat project:

   \Projects\zstack\HomeAutomation\SampleThermostat\Source

3. IAR workspace files for TemperatureSensor project:

   \Projects\zstack\HomeAutomation\SampleTemperatureSensor\<platform>

4. Source files for TemperatureSensor project:

   \Projects\zstack\HomeAutomation\SampleTemperatureSensor\Source

5. IAR workspace files for HeatingCoolingUnit project:

   \Projects\zstack\HomeAutomation\SampleHeatingCoolingUnit\<platform>

6. Source files for HeatingCoolingUnit project:

   \Projects\zstack\HomeAutomation\SampleHeatingCoolingUnit\Source

3.4.4 Additional Home Automation Project Files.

1. Source files common to all Home Automation projects:

   \Projects\zstack\HomeAutomation\Source

2. ZigBee Cluster Library source code:

   \Components\stack\zcl

3.5 Sample Light Application

3.5.1 Introduction

This sample application can be used to turn on/off the LED 4 on the device using the On/Off cluster commands, or put the device into the Identification mode (i.e., blinking the LED 4) by setting the IdentifyTime attribute to a non-zero value using the ZCL Write command.

3.5.2 Modules

The Sample Light application consists of the following modules (on top of the ZigBee stack modules):

OSAL_SampleLight.c - functions and tables for task initialization.

zcl_samplelight.c – main application function that has init and event loop function.

zcl_samplelight.h – header file for application module.

zcl_samplelight_data.c – container for declaration of attributes, clusters, simple descriptor.

3.6 Sample Switch Application

3.6.1 Introduction

This sample application can be used as the Light Switch (using the SW1) to turn on/off the LED 4 on a device running the Sample Light application.

3.6.2 Modules

The Sample Switch application consists of the following modules (on top of the ZigBee stack modules):

OSAL_SampleSw.c - functions and tables for task initialization.

zcl_samplesw.c – main application function that has init and event loop function.

zcl_samplesw.h – header file for application module.

zcl_samplesw_data.c – container for declaration of attributes, clusters, simple descriptor.

3.6.3 Sample Switch OTA Demonstration Build Configurations

3.6.3.1 Introduction

The Sample Switch Application work space contains several build configurations that demonstrate firmware upgrade using the ZigBee OTA (i.e. over the air) feature. When using OTA, to upgrade the firmware, up to two FW images may reside in the flash memory at a single time: The active image, and the downloaded image, that will be activated upon next power on.

The current image version is displayed on the LCD at initialization time.

For details of OTA update functionality see the "Z-Stack OTA Upgrade User's Guide" [6].

3.6.3.2 Modules

This build configuration adds the following modules the Sample Switch application:

hal_ota.c - OTA utility functions that are platform specific.

hal_ota.c – header file for OTA utility functions.

zcl_ota.c – event handler for base OTA ZCL event handling.

zcl_ota.h – header file for OTA even handler module.

ota_common.h – Contains definitions required when utilizing OTA.

3.7 Sample Door Lock Application

3.7.1 Introduction

This sample application can be used as a Door Lock, and can receive Door Lock cluster commands, like toggle lock/unlock, and set master PIN, from a Door Lock Controller device.

3.7.2 Modules

The Sample Door Lock application consists of the following modules (on top of the ZigBee stack modules):

OSAL_SampleDoorLock.c - functions and tables for task initialization.

zcl sampledoorlock.c – main application function that has init and event loop function.

zcl sampledoorlock.h – header file for application module.

zcl sampledoorlock_data.c – container for declaration of attributes, clusters, simple descriptor.

3.8 Sample Door Lock Controller Application

3.8.1 Introduction

This sample application can be used as a Door Lock Controller, and can send Door Lock cluster commands, like toggle lock/unlock, and set master PIN, to a Door Lock device.

3.8.2 Modules

The Sample Door Lock Controller application consists of the following modules (on top of the ZigBee stack modules):

OSAL_SampleDoorLockController.c - functions and tables for task initialization.

zcl_sampledoorlockcontroller.c – main application function that has init and event loop function.

zcl_sampledoorlockcontroller.h – header file for application module.

zcl_sampledoorlockcontroller_data.c – container for declaration of attributes, clusters, simple descriptor.

3.9 Sample Thermostat Application

3.9.1 Introduction

This sample application can be used as a Thermostat; it receives measurements of temperature from the Temperature Sensor device and sends Thermostat commands to the Heating Cooling Unit Device. Locally the heating or cooling setpoint can be configured to the Thermostat device.

3.9.2 Modules

The Sample Thermostat application consists of the following modules (on top of the ZigBee stack modules):

OSAL_SampleThermostat.c - functions and tables for task initialization.

zcl_samplethermostat.c – main application function that has init and event loop function.

zcl_samplethermostat.h – header file for application module.

zcl_samplethermostat_data.c – container for declaration of attributes, clusters, simple descriptor.

3.10 Sample Temperature Sensor Application

3.10.1 Introduction

This sample application can be used as a Temperature Sensor device to send measurements of temperature to the Thermostat device. Locally the temperature can be increased/decreased and also manually send the current temperature to the Temperature Sensor.

3.10.2 Modules

The Sample Temperature Sensor application consists of the following modules (on top of the ZigBee stack modules):

OSAL_SampleTemperatureSensor.c - functions and tables for task initialization.

zcl_sampletemperaturesensor.c – main application function that has init and event loop function.

zcl_sampletemperaturesensor.h – header file for application module.

zcl_sampletemperaturesensor_data.c – container for declaration of attributes, clusters, simple descriptor.

3.11 Sample Heating/Cooling Unit Application

3.11.1 Introduction

This sample application can be used as a Heating/Cooling Unit device to receive Thermostat commands from the Thermostat device and simulates the action of Heating/Cooling, according to the data received from the Thermostat.

3.11.2 Modules

The Sample Heating/Cooling Unit application consists of the following modules (on top of the ZigBee stack modules):

OSAL_SampleHeatingCoolingUnit.c - functions and tables for task initialization.

zcl_sampleheatingcoolingunit.c – main application function that has init and event loop function.

zcl_sampleheatingcoolingunit.h – header file for application module.

zcl_sampleheatingcoolingunit_data.c – container for declaration of attributes, clusters, simple descriptor.

3.12 Main Functions

All sample applications have the same architecture and have the same basic modules:

`<Sample_App>_Init()` – Initializes the Application task:

- Registers the application's endpoint and its simple descriptor.
- Registers the ZCL General Cluster's callbacks.
- Registers the application's attribute list.
- Register the Application to receive the unprocessed Foundation command/response messages.
- Register the EZ Mode data.
- Register for all key-press events.
- Registers a test end-point
- Registers for the following ZDO message:
    - Match_Desc_rsp

`<Sample_App>_event_loop()` – This is the "task" of the Sample Application. It handles the following events, which may vary depending on the Sample Application:

- SYS_EVENT_MSG / ZCL_OTA_CALLBACK_IND : Processes callbacks from the ZCL OTA, by calling zclSampleSw_ProcessOTAMsgs.
- SYS_EVENT_MSG / ZCL_INCOMING_MSG : Process incoming ZCL Foundation command/response messages.
- SYS_EVENT_MSG / ZDO_CB_MSG : Process response messages, by calling `<Sample_App>_ProcessZDOMsgs()`
- SYS_EVENT_MSG / KEY_CHANGE : Handle physical key-presses.
- SYS_EVENT_MSG / ZDO_STATE_CHANGE : Handle device's NWK state change.

`<Sample_App>_ProcessZDOMsgs()` – Process the following response messages:
- Match_Desc_rsp
- All other ZDO messages the Sample Application registered for.

`<Sample_App>_HandleKeys()` – Handle key-presses. Supported keys are described in [3].

`<Sample_App>_ProcessIdentifyTimeChange()` - Handle blinking of the identification LED

`<Sample_App>_IdentifyCB()` – This callback will be called by ZCL to initiate Identification. It calls `<Sample_App>_ProcessIdentifyTimeChange()` to start blinking the respective LED.

`<Sample_App>_ProcessIncomingMsg()` - This is a stub function, providing the developer with an easy starting point for handling ZCL Foundation incoming messages.

`<Sample_App>_EZModeCB()` – This callback will be informed of EZ Mode events and status.

4. Using the sample applications as base for new applications

The HA Sample Applications are intended to be used as foundations for the user's applications. Modifying them will usually consist of the following steps:

1. Copy the Sample Application of your choice to a new folder, and name it according to your new application:

   Copy

   Project\zstack\HomeAutomation\SampleLight*.*

   To

   Project\zstack\HomeAutomation\YourApp*.*
2. Rename the files to match your application's name:

   Under Projects\zstack\HomeAutomation\YourApp\Source, rename the files as follows:
   - OSAL_SampleLight.c → OSAL_YourApp.c
   - zcl_samplelight.c → zcl_yourapp.c

   Etc.
3. In Projects\zstack\HomeAutomation\YourApp\CC2538, rename the following files:
   - SampleLight.ewd→YourApp.ewd
   - SampleLight.ewp→YourApp.ewp
   - SampleLight.eww→YourApp.eww

   Delete all the other files and subfolders from this folder, if there are any.

4. Using a text editor, replace all occurrences of "SampleLight" (case insensitive) to "YourApp", in YourApp.ewp and YourApp.eww.

5. You can now open YourApp.eww with IAR. Notice that the application files that are included in the project are the files from your application:

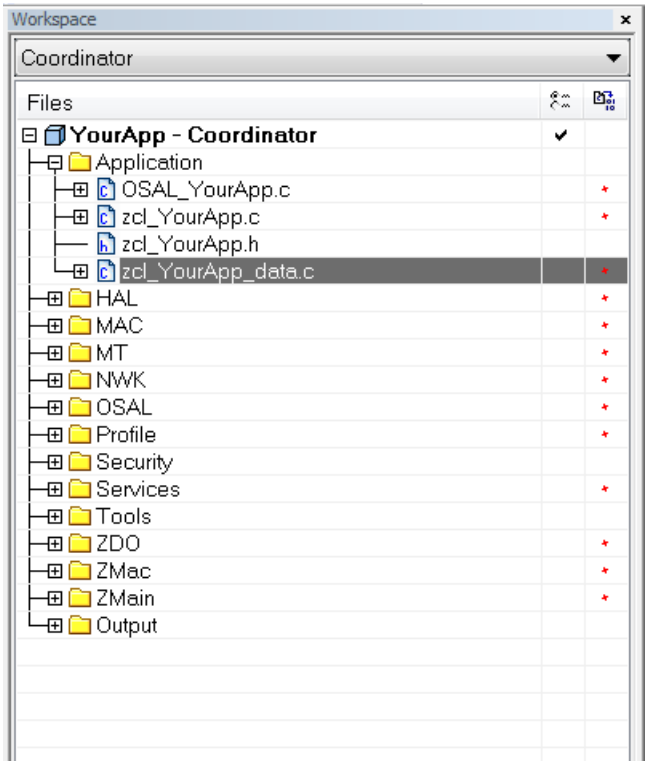

Figure 1 - Application files within YourApp

6. You will have to edit the application files, to replace any occurrence of "zcl_samplelight.h" with "zcl_yourapp.h".

7. Now you are ready to manipulate the application code to suite your needs, e.g. modify key-press activities, change the device type, modify the supported clusters, etc. For further information, please refer to the applicable documents, listed at the end of this user guide.

5. Compilation Flags

5.1 Mandatory Compilation Flags

The following compilation flags are mandatory across all devices. Compilation flags (also called options) can be enabled using a –D in the f8wConfig.cfg, or in the compiler's predefined constants section (also called command-line). To disable the option, either put a // in front of the –D, or put a small "x" in front of the compiler's predefined constant in the IAR Embedded Workbench preprocessor section of the project (*e.g. xNV_RESTORE).*

These compilation flags are defined in the preprocessor section on each sample application IAR project:

- `SECURE=1`
- `TC_LINKKEY_JOIN`

All applications include these clusters and are defined in the IAR Workbench preprocessor section:

- `ZCL_READ`
- `ZCL_WRITE`
- `ZCL_REPORT`
- `ZCL_EZMODE`
- `ZCL_BASIC`
- `ZCL_IDENTIFY`

A description of the flags and commands for the specific cluster can be found in f8wZCL.cfg file.

Note that in IAR Embedded Workbench for 8051, the definitions in the project's preprocessor defined symbols list will override definitions set in the file *f8wConfig.cfg*. In IAR Embedded Workbench for ARM (CC2538) works the opposite way, the definitions in *f8wConfig.cfg* file will override the project's preprocessor defined symbols.

5.2 Mandatory Compilation Flags Per Device

The sample applications each have their own mandatory flags. Note that enabling a cluster will enable both the Send and ProcessIn functions. It is up to the application to decide which side to use (for example, the SampleSwitch uses the Send side of the ZCL_ON_OFF cluster, whereas the SampleLight uses the ProcessIn side of the ZCL_ON_OFF Cluster).

SampleLight:

- `ZCL_GROUPS`
- `ZCL_SCENES`
- `ZCL_ON_OFF`

SampleSwitch:

- `ZCL_ON_OFF`

SampleDoorLock:

- `ZCL_GROUPS`
- `ZCL_SCENES`
- `ZCL_DOORLOCK`

SampleDoorLockController:

- ZCL_GROUPS
- ZCL_SCENES
- ZCL_DOORLOCK

SampleThermostat:

- ZCL_ON_OFF
- ZCL_HVAC_CLUSTER

SampleTemperatureSensor:

- ZCL_TEMPERATURE_MEASUREMENT

SampleHeatingCoolingUnit:

- ZCL_ON_OFF
- ZCL_HVAC_CLUSTER

5.3 Optional Compilation Flags

The following flags may be included in a Home Automation project's options preprocessor defined symbols list, each one adding code to enable different feature.

The following flags are set by default on all the sample projects (unless otherwise specified):

- HOLD_AUTO_START – prevents the device from automatically trying to join a network after power-up when Factory New, and causes it to wait for an external event instead. This is enabled for ZEDs and ZRs by default. It is disabled for ZCs by default (that is, ZigBee Coordinators starts automatically).

The following flags are set by default on all the SmartRF05EB projects:

- MT_TASK – enables any MT functionality, via serial link.
- MT_APP_FUNC – enables MT_APP functionality for communication with a host processor.
- MT_SYS_FUNC – enables interaction at system level, required for communicating with Z-Tool.
- ZTOOL_P1 – required for communicating with serial interface tool.

Additional flags:

- MT_UTIL_FUNC – enables MT_UTIL functionality, e.g. key press events emulation.
- MT_ZDO_FUNC – enables MT_ZDO functionality, e.g. sending *Mgmt_Permit_Joining_req* command.
- POWER_SAVING – enables an end-device to sleep and conserve power, when inactive. Should be defined on any battery-powered device.
- ZCL_LEVEL_CTRL – enables the Level Control cluster. Required to support dimming capabilities.
- HAL_PWM – enables the PWM (pulse width modulation) functionality in the CC2538 GP Timer hardware subsystem. Note, enabling this flag will drive LED 1 with PWM output which is used to observe "level" values for the Sample Light.
- HAL_LED=FALSE – disables the normal LED functionality on the SmartRF06. This flag must be used in conjunction with HAL_PWM to properly observe level or dimming behavior on the SmartRF06 LED 1.

6. Clusters, Commands and Attributes

Each application supports a certain number of clusters. Think of a cluster as an object containing both methods (commands) and data (attributes).

Each cluster may have zero or more commands. Commands are further divided into Server and Client-side commands. Commands cause action, or generates a response.

Each cluster may have zero or more attributes. All of the attributes can be found in the *zcl_sampleapp_data.c* file, where "sampleapp" is replaced with the given sample application (e.g. *samplesw_data.c* for the sample on/off light switch). Attributes describe the current state of the device, or provide information about the device, such as whether a light is currently on or off.

All clusters and attributes are defined either in the ZigBee Cluster Library specification, or the ZigBee Home Automation Specification.

6.1 Attributes

Attributes are found in a single list called `zclSampleApp_Attrs[ ]`, in the zcl_sampleapp_data.c file. Each attribute entry is initialized to a type and value, and contains a pointer to the attribute data. Attribute data types can be found in the ZigBee Cluster Library.

The attributes must be registered using the `zcl_registerAttrList ( )` function during application initialization, one per application endpoint.

Each attribute has a data type, as defined by ZigBee (such as `UINT8`, `INT32`, etc…). Each attribute record contains an attribute type and a pointer to the actual data for the attribute. Read-only data can be shared across endpoints. Data that is unique to an endpoint (such as the OnOff attribute state of the light) should have a unique C variable.

All attributes can be read. Some attributes can be written. Some attributes are reportable (can be automatically sent to a destination based on time or change in attribute). Some attributes are saved as part of a "scene" that can later be recalled to set the device to a particular state (such as a light on or off). The attribute access is controlled through a field in the attribute structure.

To store an attribute in non-volatile memory (to be preserved across reboots) refer to **section 8.2**.

6.2 Adding an Attribute Example

To add an additional attribute to a project, refer to the attribute's information within the Home Automation Specification [1]. Using the DoorLock cluster as an example, the following will show how to add the "Max PIN Code Length" attribute to the DoorLock project. This process can be replicated across all Home Automation projects.

All attributes in use by an application are defined within the project source file's zcl_sampleapplication_data.c file. For this DoorLock example, this data file is: *zcl_sampledoorlock_data.c*. Locate the section defined as *Attribute Definitions* and include the "Max PIN Code Length" attribute using the format:

```
1  {
2      ZCL_CLUSTER_ID_CLOSURES_DOOR_LOCK,
3      { // Attribute record
4        ATTRID_DOORLOCK_NUM_OF_MAX_PIN_LENGTH,
5        ZCL_DATATYPE_UINT8,
6        ACCESS_CONTROL_READ,
7        (void *)&zclSampleDoorLock_NumOfMaxPINLength
8      }
```



```
9     },
```

Figure 2 - Example of Number of Max PIN Length Attribute Record

Line 2 represents the cluster ID, line 4 represents the attribute ID, line 5 the data type, line 6 the read/write attribute, and line 7 the pointer to the variable used within the application.

The cluster ID can be retrieved from the *zcl.h* file, the attribute ID can be found within the (in this case) *zcl_closures.h* file, and the remaining information from the Home Automation Specification [1].

By including the attribute within this list, devices are able to interact with the attributes on other devices. This addition in the attribute list must be reflected in the `SAMPLEDOORLOCK_MAX_ATTRIBUTES` macro in the *zcl_sampledoorlock.h* file. Also within that file, define the external variable using proper coding conventions:

```
extern uint8 zclSampleDoorLock_NumOfMaxPinLength
```

Finally, define the variable within *zcl_sampledoorlock.c* to be used by the application. Note the default value and valid range of the variable in the specification.

6.3 Initializing Clusters

For the application to interact with a cluster, the cluster's compile flag must be enabled (if applicable to the cluster) in the project's configuration and the cluster's source file must be added to the project's Profile folder within the IAR Workspace. An example of this can be seen in Figure 3 for the SampleDoorLock app. See *f8wZCL.cfg* for a list of cluster compile flags.

Once enabled, the cluster's callbacks can be registered within the application (refer to **section 6.5**).

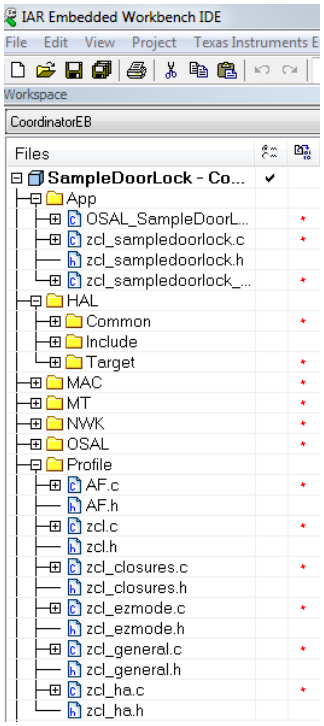

Figure 3 - List of Cluster Source Files in Profile Folder

6.4 Cluster Architecture

All clusters follow the same architecture.

The clusters take care of converting the structures passed from native format to over-the-air format, as required by ZigBee. All application interaction with clusters takes place in native format.

They all have the following functions:

- Send – This group of commands allows various commands to be send on a cluster
- ProcessIn – This function processes incoming commands.

There is usually one send function for each command. The send function has either a set of parameters or a specific structure for the command.

If the application has registered callback functions, then the ProcessIn will direct the command (after it's converted to native form) to the application callback for that command.

6.5 Cluster Callbacks Example

Callbacks are used so that the application can perform the expected behavior on a given incoming cluster command. It is up to the application to send a response as appropriate. Z-Stack provides the parsing, but it is up to the application to perform the work.

A cluster's callback functions are registered within the application's initialization function by including the application's endpoint and a pointer to the callback record within a register commands callback function. Figure 4 shows an example of the general cluster's callback record list. The commands are registered to their respective callback functions as defined within the cluster's profile.

As an example, once a BasicReset command reaches the application layer on a device, the cluster's callback record list points the command to the BasicReset callback function: `zclSampleDoorLockController_BasicResetCB`. The application reset command can then reset all data back to Factory New defaults.

The callback function in an application provides additional processing of a command that is specific to that application. These callback functions work alongside the response to the incoming command, if a response is appropriate.

```
/*********************************************************************
 * ZCL General Profile Callback table
 */
static zclGeneral_AppCallbacks_t zclSampleDoorLockController_CmdCallbacks =
{
  zclSampleDoorLockController_BasicResetCB,             // Basic Cluster Reset command
  zclSampleDoorLockController_IdentifyCB,               // Identify command
  NULL,                                           // Identify Trigger Effect command
  zclSampleDoorLockController_IdentifyQueryRspCB,       // Identify Query Response command
  NULL,                                           // On/Off cluster commands
  NULL,                                           // On/Off cluster enhanced command Off with Effect
  NULL,                                           // On/Off cluster enhanced command On with Recall Global Scene
  NULL,                                           // On/Off cluster enhanced command On with Timed Off
  NULL,                                           // Level Control Move to Level command
  NULL,                                           // Level Control Move command
  NULL,                                           // Level Control Step command
  NULL,                                           // Level Control Stop command
  NULL,                                           // Group Response commands
  NULL,                                           // Scene Store Request command
  NULL,                                           // Scene Recall Request command
  NULL,                                           // Scene Response command
  NULL,                                           // Alarm (Response) commands
  NULL,                                           // RSSI Location command
  NULL                                            // RSSI Location Response command
};
```

Figure 4 - Cluster Callbacks Example

7. EZ-Mode

EZ-Mode provides an ability to easily bind (connect) two devices together for normal communication, whether the devices are currently on a ZigBee network or not. It includes both the ability for network steering, and finding and binding. EZ-Mode includes the following features:

- Network Steering – finds the first open network
- Finding and Binding – finds a remote node (anywhere in the network) and initiates appropriate bindings based on an application supplied active output cluster list.
- Autoclose - for rapid binding of many pairs of devices
- Application Callbacks – allows the application to present an appropriate user interface
- Any Invoke Method – applications can be creative in the method they invoke EZ-Mode

As an example, EZ-Mode allows light switches to be bound to lights, or a temperature sensor or heating cooling unit to a thermostat.

EZ-Mode is smart enough to know that a light switch cannot be bound to a temperature sensor, or to another light switch.

In the sample applications, the EZ-Mode invoke call is performed in the application HandleKeys function (*e.g.* `zclSampleThermostat_HandleKeys`), on switch 2 for all Home Automation Sample Applications. Since ZRs and ZEDs do not autostart, this also puts the node on the network, then, if appropriate, will bind to the other remote device.

Devices that are on the network will open up the network (enable joining) while EZ-Mode is in effect. EZ-Mode will close the network as soon as both nodes in the pair have found each other and determined their bindings.

There is only 1 EZ-Mode state machine per device. The EZ-Mode Invoke function (`zcl_InvokeEZMode`) is a toggle. That is, if EZ-Mode has already been started, invoke will cancel EZ-Mode on that device.

The following terms should be understood to understand EZ-Mode:

- Invoke – Initiate EZ-Mode on a device
- Opener – A node that is already on a network when EZ-Mode is invoked
- Joiner – A node that is not on a network when EZ-Mode is invoked
- Initiator – A node that initiates transactions (client) on the active clusters
- Target – A node that receives transactions (server) from active clusters
- Active clusters – The main functionality of a device (e.g OnOff for a light switch, or DoorLock for a DoorLockController)

If NV_RESTORE is enabled, network configuration is stored in NV memory, EZ-Mode has no method for removing bindings directly, other than to factory reset a device. The Home Automation sample applications reset NV memory on Shift-SW5 (i.e. press and hold Button 1 and then press down the Joystick), this functionality is only available in SmartRF05 + CC2530. To reset NV memory to factory defaults in SmartRF06 + CC2538, the device should be reprogrammed.

While more sophisticated commissioning can be performed by various over-the-air ZigBee calls, EZ-Mode provides a solid base-line for connecting the appropriate devices together.

7.1 EZ-Mode Interface

The EZ-Mode interface can be found in *zcl_ezmode.h*. The EZ-Mode application interface is performed through 3 functions and a single application callback:

- `zcl_RegisterEZMode` – Must be called before EZ-Mode can be used

- `zcl_InvokeEZMode` – Used to invoke or cancel EZ-Mode
- `zcl_EZModeAction` – Used by the application to inform EZ-Mode of various events (such as when joining the network or receiving and IdentifyQuery command).

The application must support the Identify Cluster. EZ-Mode will continue to identify itself for `EZMODE_TIME` (which defaults to 3 minutes).

7.2 EZ-Mode Diagrams

EZ-Mode flows through the following states:

- Forming/Joining the network (if not yet on a network)
- Opening the network (through Permit Joining Request)
- Identifying Itself
- Finding a remote node in EZ-Mode
- Matching active output clusters and binding them
- AutoClose after a small period of time to allow Matching in both directions
- Finish up with status code

Network steering and Finding and Binding are shown in the following two flowchart diagrams.

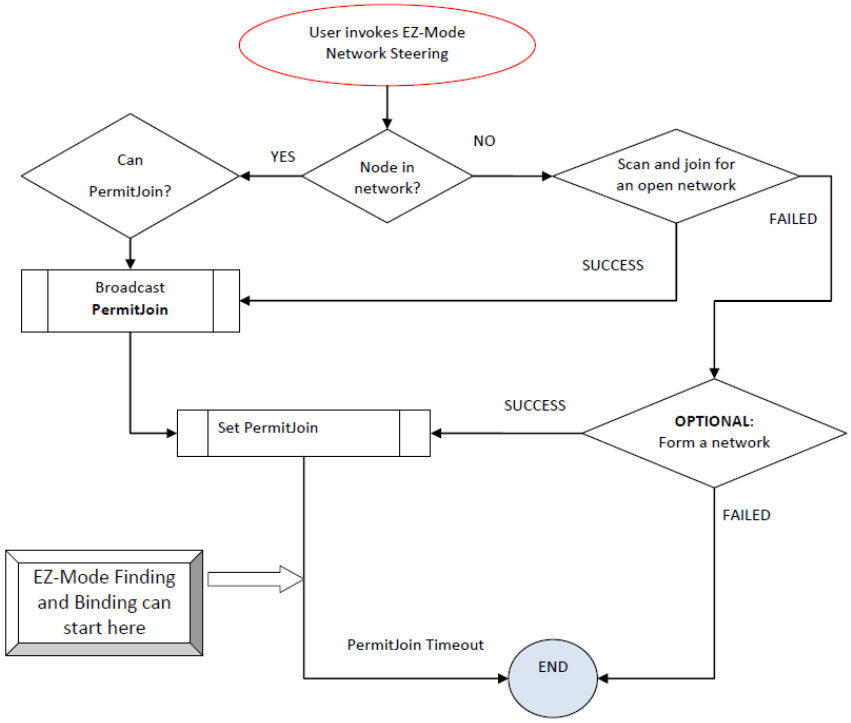

Figure 5 - EZ-Mode Network Steering Flowchart

The Network Steering Flowchart indicates how EZ-Mode forms, joins, opens and closes the network. It is assumes that only 1 open network is in the area on the list of HA channels. A network's normal state is closed.

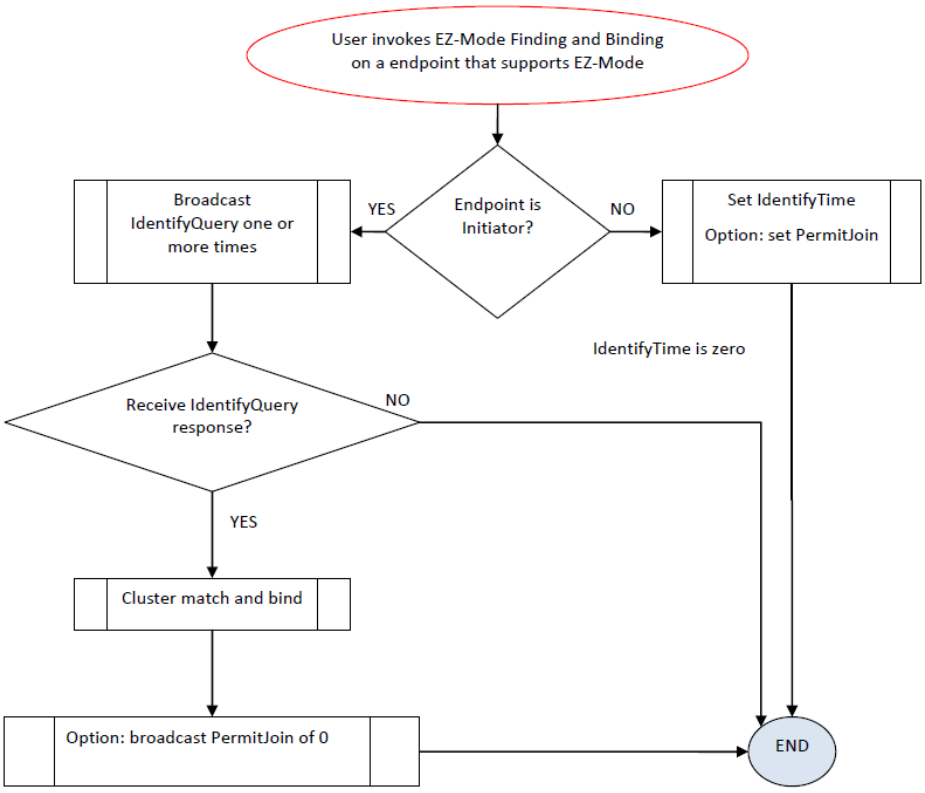

Figure 6 – EZ-Mode Finding and Binding Flowchart

The Finding and Binding Flowchart shows how EZ-Mode finds and matches to another device in the network. The devices can be many hops away (using mesh networking). See the HA specification for which devices are initiators and which ones are targets.

7.3 EZ-Mode Code

The bulk of the code for EZ-Mode is found in *zcl_ezmode.c*. This includes the full state machine and the user interface. In addition, the Application must supply some functionality to enable EZ-Mode to transition through various states.

The application must supply:

- Notification of forming/joining
- Notification of cluster matching
- Enter/exit identify mode
- 2 Events (one for state changes, one for an overall EZ-Mode timeout)

The following example shows how the SampleLight application informs EZ-Mode to the overall timeout, and when it's time to process the next state.

```
#ifdef ZCL_EZMODE
  if ( events & SAMPLELIGHT_EZMODE_NEXTSTATE_EVT )
  {
    zcl_EZModeAction ( EZMODE_ACTION_PROCESS, NULL );
    return ( events ^ SAMPLELIGHT_EZMODE_NEXTSTATE_EVT );
  }

  if ( events & SAMPLELIGHT_EZMODE_TIMEOUT_EVT )
  {
```



```
      zcl_EZModeAction ( EZMODE_ACTION_TIMED_OUT, NULL ); // EZ-Mode timed out
      return ( events ^ SAMPLELIGHT_EZMODE_TIMEOUT_EVT );
    }
#endif // ZCL_EZMODE
```

EZ-Mode may be enabled or disabled by the use of the `ZCL_EZMODE` flag. EZ-Mode requires the Identify cluster, so make sure to also enable `ZCL_IDENTIFY`.

If EZ-Mode is disabled, the sample applications will use End Device Binding to bind applications together. If EZ-Mode is disabled, make sure to also enable auto start by disabling the `HOLD_AUTO_START` flag in the Compiler's predefined compile-time flags, or the node won't join the network.

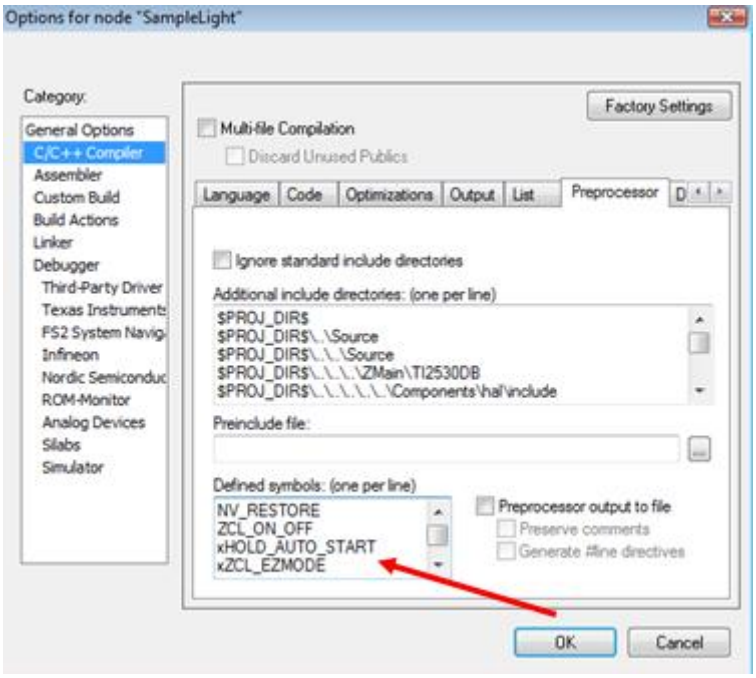

Figure 7 – To Disable EZ-Mode, disable both ZCL_EZMODE and HOLD_AUTO_START

To disable an option, place an 'x' in front of the name. This way, the option can later be enabled by removing the 'x'. (e.g. `xHOLD_AUTO_START` means hold auto start is disabled).

8. Additional Information for HA Applications

The sample applications implement a minimal set of features to operate. Optional features, as described in the ZigBee Home Automation Specification [1], can be added for a more full-featured application.

In addition to enabling the appropriate clusters and attributes, application must also select a ZigBee device type and enable non-volatile memory (NVM).

8.1 ZigBee Device Type

ZigBee allows for 3 device types: a ZigBee Coordinator, ZigBee Router, and ZigBee End Device. It must be decided for each application which ZigBee device type the application will be. The following are basic guidelines. For complete details, please see [4].

Use a ZC for device that will start the network, typically a base-station

Use a ZR for devices that will be mains powered, such as an outlet. Only use this type if there won't be very many of these devices at a given facility (12 or less in radio range of each other). Otherwise, use ZED.

Use a ZED for devices that will normally sleep in order to conserve power (usually battery operated devices), or for mains-power operated devices, if there are many such devices in close proximity (see above).

8.2 Storing Variables to Non-Volatile Memory

In general, a device must have non-volatile memory enabled to be certified, because it must remember its network configuration.

To save a variable to NVM, an item ID must be created for that variable. IDs reserved for applications range from 0x0401 to 0x0FFF. For a complete list of these IDs, refer to the *ZComDef.h* file.

Once the item ID is defined, it must be initialized using `osal_nv_item_init()`. This function passes the item ID, length of the variable to be stored, and the variable (if the user wants to use the initial value).

To write an item to NVM, use `osal_nv_write()`. This function passes the item ID, the index offset into an item, the length of data to write, and the variable to write.

To read the item from NVM, use `osal_nv_read()`. This function passes the item ID, the index offset into an item, length of data to read, and the variable to read the data to. These functions can be found in the *OSAL_Nv.c* file.

8.3 User Interface

The following user interface elements should be in each application:

- Some way to invoke EZ-Mode on each endpoint
- Some way to reset the entire device to factory new settings (not on a network, etc…)
- Some way to cause the application action (e.g. a switch turning on a light)
- Set up battery powered devices as sleepy devices

[i] The next hop device may or may not be the destination device of a data request. Therefore, the AF data confirmation of delivery to the next hop must not be misinterpreted as confirmation that the data request was delivered to the destination device. Refer to the discussion of delivery options (`AF_ACK_REQUEST` in the Z-Stack API).

[ii] See sections 1.2.3 and 1.2.6.1.3.

[iii] For a complete list of all of the types of SYS_EVENT_MSG messages, refer to "Global System Messages" in ZComDef.h.

[iv] See "MAC status values" in ZComDef.h.

