Z-Stack Home TI-RTOS Developer's Guide

Texas Instruments, Inc.
San Diego, California USA

Version	Description	Date
1.0	Initial release	02/20/2015

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
TIRTOS	Texas Instruments Real Time Operating System
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, Z-Stack TI-RTOS Developer's Guide.

[5] Z-Stack TI-RTOS API

2. Overview

2.1 Introduction

This document refers to Z-Stack™ Home Sample Applications for CC2630 and CC2650 platforms, using the SimpleLink TI-RTOS. 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].

All sample applications utilize the Z-Stack API [5] for communication with Z-Stack Thread and perform registration of application elements (Endpoints, Callbacks, ZDO commands, etc.) and Z-Stack configuration. In addition, every sample application makes use LED, KEYs and LCD functionality by accessing the application helper modules included at the application layer. 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 could instantiate multiple Endpoints; 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.

2.1.1 Application Design

The Application Image runs in one thread and Z-Stack runs in a separate thread, as shown in Figure 1.

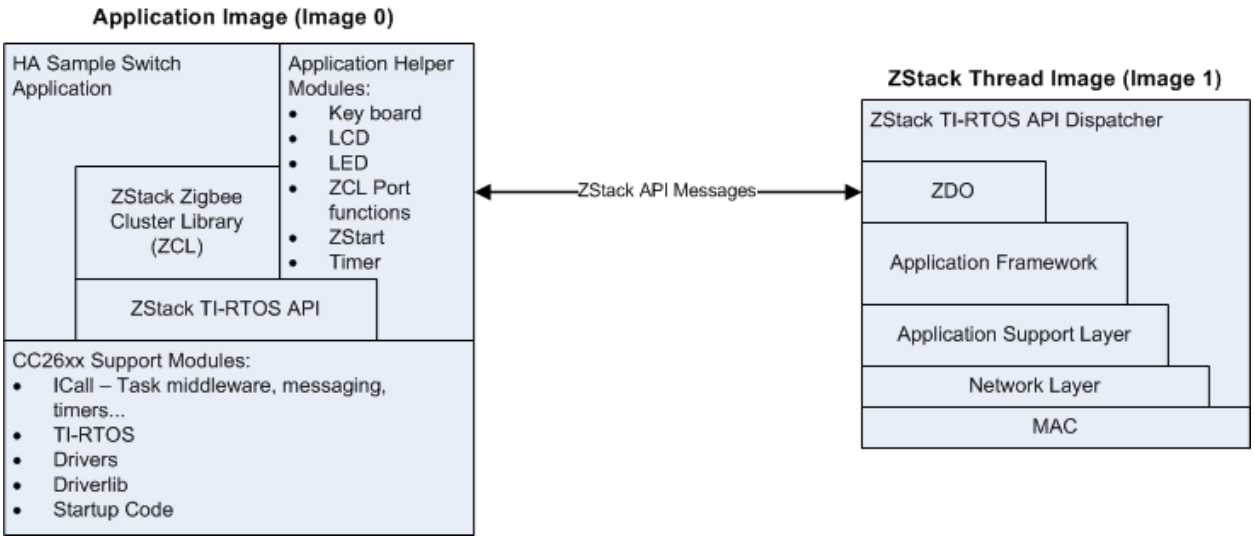

Figure 1 - Z-Stack/Application architecture

The application communicates with Z-Stack using the functions defined in `zstackapi.c` and ICall messages. For the purpose of this document *SampleDoorLock* application will be used as an example.

2.1.2 Mandatory Methods

The application task takes care of initializing all Z-Stack parameters and also registering all the necessary elements for the application to operate.

2.1.2.1 Application Initialization

In the sample applications, the application's entry point is named according to the following pattern: <Application-Name>_task (e.g. `DLSApp_task()`). The application's entry point function should accomplish the following:

- Save and register the function pointers to the NV drivers `zclport_registerNV()`, for application data required to be stored in NV.
- Initialize application `DLSApp_initialization()`:
    - Register EZMode callback function
    - Setup application timers
    - Board key, LCD and LED setup
    - Register application with ICall to send and receive messages
    - Initialize ZStack: register eindpoints, setup CB for indications from the Stack, register for EZMode, start the Z-Stack thread, register attribute list and command callbacks.
- Kick start the application processing task `DLSApp_process()`, in charge of processing all events.

2.1.2.2 Application Processing Task Handler

In the sample applications, the function to perform task event handling is named according to the following pattern: <Application-Name>_process (e.g. `DLSApp_process()`). This function waits on a signal to the semaphore associated with the application thread; the signal can be either a queued message or an event posted to the semaphore.

2.1.3 Z-Stack Message Handler

Messages coming from Z-Stack are handled by `DLSApp_processZStackMsgs()`. The full list of messages can be found in `zstackmsg.h`. Some of the messages used by sample applications are:

- `zstackmsg_CmdIDs_DEV_STATE_CHANGE_IND`
- `zstackmsg_CmdIDs_ZDO_MATCH_DESC_RSP`
- `zstackmsg_CmdIDs_AF_INCOMING_MSG_IND`

There are other messages that can be processed by the sample application and a handler has to be implemented if the application needs to process them:

- `zstackmsg_CmdIDs_AF_DATA_CONFIRM_IND`
- `zstackmsg_CmdIDs_ZDO_DEVICE_ANNOUNCE`
- `zstackmsg_CmdIDs_ZDO_NWK_ADDR_RSP`
- `zstackmsg_CmdIDs_ZDO_IEEE_ADDR_RSP`
- `zstackmsg_CmdIDs_ZDO_NODE_DESC_RSP`
- `zstackmsg_CmdIDs_ZDO_POWER_DESC_RSP`
- `zstackmsg_CmdIDs_ZDO_SIMPLE_DESC_RSP`
- `zstackmsg_CmdIDs_ZDO_ACTIVE_EP_RSP`
- `zstackmsg_CmdIDs_ZDO_COMPLEX_DESC_RSP`
- `zstackmsg_CmdIDs_ZDO_USER_DESC_RSP`
- `zstackmsg_CmdIDs_ZDO_USER_DESC_SET_RSP`
- `zstackmsg_CmdIDs_ZDO_SERVER_DISC_RSP`
- `zstackmsg_CmdIDs_ZDO_END_DEVICE_BIND_RSP`
- `zstackmsg_CmdIDs_ZDO_BIND_RSP`
- `zstackmsg_CmdIDs_ZDO_UNBIND_RSP`
- `zstackmsg_CmdIDs_ZDO_MGMT_NWK_DISC_RSP`
- `zstackmsg_CmdIDs_ZDO_MGMT_LQI_RSP`
- `zstackmsg_CmdIDs_ZDO_MGMT_RTG_RSP`
- `zstackmsg_CmdIDs_ZDO_MGMT_BIND_RSP`

- zstackmsg_CmdIDs_ZDO_MGMT_LEAVE_RSP
- zstackmsg_CmdIDs_ZDO_MGMT_DIRECT_JOIN_RSP
- zstackmsg_CmdIDs_ZDO_MGMT_PERMIT_JOIN_RSP
- zstackmsg_CmdIDs_ZDO_MGMT_NWK_UPDATE_NOTIFY
- zstackmsg_CmdIDs_ZDO_SRC_RTG_IND
- zstackmsg_CmdIDs_ZDO_CONCENTRATOR_IND
- zstackmsg_CmdIDs_ZDO_NWK_DISC_CNF
- zstackmsg_CmdIDs_ZDO_BEACON_NOTIFY_IND
- zstackmsg_CmdIDs_ZDO_JOIN_CNF
- zstackmsg_CmdIDs_ZDO_LEAVE_CNF
- zstackmsg_CmdIDs_ZDO_LEAVE_IND
- zstackmsg_CmdIDs_SYS_RESET_IND
- zstackmsg_CmdIDs_AF_REFLECT_ERROR_IND
- zstackmsg_CmdIDs_ZDO_TC_DEVICE_IND
- zstackmsg_CmdIDs_DEV_PERMIT_JOIN_IND

2.1.4 Event Handler

The events that are processed by the application are:

- DLSAPP_KEY_EVENT
- DLSAPP_IDENTIFY_TIMEOUT_EVT
- DLSAPP_MAIN_SCREEN_EVT
- DLSAPP_EZMODE_NEXTSTATE_EVT
- DLSAPP_EZMODE_TIMEOUT_EVT

Each application can define new events according to the specific needs of the application.

2.2 Application Configuration

Z-Stack configuration parameters can be found in znwk_config.h. A sample application compiled as an End-Device will try to join a network on one of the channels specified by ZNWK_DEFAULT_CHANLIST. If ZNWK_CONFIG_PAN_ID is not defined as 0xFFFF, the End- Device will be constrained to join only the Pan ID thusly defined.

2.2.1 Auto/Hold Start

An application can control if the 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 the application should control that from <Application-Name>_initializeStack() by sending a request to the stack with the proper delay value by calling Zstackapi_DevStartReq(). All Home Automation sample applications hold to join a network until the EZ Mode process is triggered by user intervention, see [3] for sample application use.

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 is enabled by default in the devices, see [4] to disable saving Network information into Non-volatile memory in Z-Stack.

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 zstackmsg_CmdIDs_DEV_STATE_CHANGE_IND message. The zstack_DevState enum values can be found in *zstack.h*.

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 "*SampleDoorlock*" application. provided in the Z-Stack Home installer. Functions described in this section can be seen in more detail in [5].

2.3.1 Initialization

During system power-up and initialization, the applications's initialization function will be invoked (see 2.1.2.1). The structure of this function is explained below:

2.3.1.1 ZCL EZMode callback registration

```
zclport_registerEZModeTimerCB(DLSApp_changeEZModeTimerCallback);
```

Register the callback function to control EZ mode timers. For more details on this function see [5].

2.3.1.2 Timer

```
DLSApp_initializeClocks();
```

Initializes all the timers used by the application. It configures timeout values and callback functions associates with a specific timer.

2.3.1.3 Peripherals setup

```
Board_Key_initialize(DLSApp_changeKeyCallback);
```

Setup key functionality and register key pressing call back function, as well as timer definition for KEY functionality.

```
Board_LCD_open();
```

LCD setup and initialization. For more details on this function see [5].

```
Board_Led_initialize();
```

LED setup and initialization. For more details on this function see [5].

2.3.1.4 Application Registration

```
ICall_registerApp(&zdlEntity, &sem);
```

The application thread is registered as and ICall dispatcher so the application can send and receive messages to Z-Stack thread. For more details on this function see [5].

2.3.1.5 Z-Stack initialization

```
DLSApp_initializeStack();
```

- The application registers the Endpoints used by the application
  `DLSApp_registerEndpoints();`
- Set up call back ZDO indications required by the application
  `DLSApp_setupZStackCallbacks();`
- Sends the request to start the device to form or join a network
  `Zstackapi_DevStartReq(zdlEntity, &startReq);`
- Register ZCL General Cluster Library callback functions
  `zclGeneral_RegisterCmdCallbacks(DLSAPP_EP, &cmdCallbacks);`
- Register specific ZCL cluster (Doorlock for this example) Library function
  `zclClosures_RegisterDoorLockCmdCallbacks(DLSAPP_EP, &doorLockCmdCallbacks);`
- Register the application's attribute list


```
zcl_registerAttrList(DLSAPP_EP, DLSAPP_MAX_ATTRIBUTES, zdlAttrs);
```

- Update the Z-Stack configuration parameters
  `DLSApp_writeParameters();`

For more details on these function see [5].

2.3.2 Event Processing

Whenever a Z-Stack message is received, the event processing function
`DLSApp_processZStackMsgs(pMsg)`

will be invoked in turn from the `DLSApp_process(void)` task processing loop.

The parameter to the task event handler is a pointer to the incoming Z-Stack message. One event is received per Z-Stack message.

The handler function identifies the type of event:

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

And handles it according to application specific needs, some of the events used by the sample applications are:

```
case zstackmsg_CmdIDs_DEV_STATE_CHANGE_IND:
```

Whenever the network state changes, the application is notified with this event message. The indication contains the state the device is, either a Coordinator, Router or End-Device.

```
if( (pInd->req.state == zstack_DevState_DEV_ZB_COORD)
     || (pInd->req.state == zstack_DevState_DEV_ROUTER)
     || (pInd->req.state == zstack_DevState_DEV_END_DEVICE) )
```

If an over the air MatchDecriptorResponse message is received, the application handles it:

```
case zstackmsg_CmdIDs_ZDO_MATCH_DESC_RSP:
```

and give it to ZCL EZMode module to process.

For handling of ZCL messages the event handler uses:

```
case zstackmsg_CmdIDs_AF_INCOMING_MSG_IND:
```

The incoming message is parsed and passed to ZCL processing function

```
zcl_ProcessMessageMSG(&afMsg).
```

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 follow the application configuration and initialization elements described above.

As part of the application 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 elements.

- Register the *simple descriptor* (e.g. *DoorLock*) with the HA Profile using `DLSApp_registerEndpoints(void)`.
- Register the *command callbacks table* (e.g. `cmdCallbacks`) with the General functional domain using `zclGeneral_RegisterCmdCallbacks(DLSAPP_EP, &cmdCallbacks)`.
- Register the *application specific* (e.g. `doorLockCmdCallbacks`) *command callbacks table* using `zclClosures_RegisterDoorLockCmdCallbacks(DLSAPP_EP, &doorLockCmdCallbacks)`.
- Register the *attribute list* (e.g. `zdlAttrs`) with the ZCL Foundation layer using `zcl_registerAttrList(DLSAPP_EP, DLSAPP_MAX_ATTRIBUTES, zdlAttrs)`.

3.3 Software Architecture

The application receives incoming data messages from Z-Stack and are processed by

```
case zstackmsg_CmdIDs_AF_INCOMING_MSG_IND:
{
  // Process incoming data messages
  zstackmsg_afIncomingMsgInd_t *pInd
                    = (zstackmsg_afIncomingMsgInd_t *)pMsg;
  DLSApp_processAfIncomingMsgInd( &(pInd->req) );
}
break;
```

The messages are then passed to the ZCL message processor, which will identify what type of message has been received, either foundation message or profile/cluster specific message, and send it to the appropriate processing callback function, if registered.

Applications receive data in callback functions after they register the callbacks. Each cluster has its own register function and set of callback functions (*e.g.* `doorLockCmdCallbacks`).

A cluster's callback functions are registered within the application's initialization function 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. DLSApp_ BasicReset)*, 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.

Applications send data through various send functions (*e.g.* `zclGeneral_SendOnOff_CmdToggle`).

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 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 DoorLock Sample Application (DoorLock and DoorLockController).

1. IAR workspace and Source files for DoorLock project:

   \Projects\zstack\HomeAutomation\SampleDoorLock\CC26xx

2. IAR workspace and Source files for DoorLockController project:

   \Projects\zstack\HomeAutomation\SampleDoorLockController\CC26xx

3.4.2 Switch Sample Application.

1. IAR workspace and Source files for Switch project:

   \Projects\zstack\HomeAutomation\SampleSwitch\CC26xx

3.4.3 TemperatureSensor Sample Application.

1. IAR workspace and Source files for TemperatureSensor project:

   \Projects\zstack\HomeAutomation\SampleTemperatureSensor\CC26xx

3.4.4 SensorTag Sample Application.

1. IAR workspace and Source files for SensorTag project:

   \Projects\zstack\HomeAutomation\SensorTag\CC2650

3.4.5 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. Peripheral utility source code:

   \Projects\zstack\common\CC26xx\SmartRF06

   \Projects\zstack\common\CC26xx

3.4.6 Common Application Project Files.

Home Automation sample applications have a set of common files in:

\Projects\zstack\common\CC26xx\

These files are:

- *board_key.c*
- *board_key.h*
- *board_lcd.c*
- *board_lcd.h*
- *board_led.c*
- *board_led.h*
- *util.c*
- *util.h*
- *zcl_port.c*

- *zcl_port.h*

For a description of these files see [5].

3.5 Sample Door Lock Application

3.5.1 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.

3.5.2 Modules

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

dlsapp.c – main application module that has the declaration of attributes, clusters, simple descriptor, initialization and process functions.

dlsapp.h – header file for application module.

3.6 Sample Door Lock Controller Application

3.6.1 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.

3.6.2 Modules

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

dlcsapp.c – main application module that has the declaration of attributes, clusters, simple descriptor, initialization and process functions.

dlscapp.h – header file for application module.

3.7 Sample Switch Application

3.7.1 Introduction

This sample application can be used as the Light Switch (using the SW1) to turn on/off (toggle) a remote Light.

3.7.2 Modules

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

switch.c – main application module that has the declaration of attributes, clusters, simple descriptor, initialization and process functions.

switch.h – header file for application module.

3.8 Sample Temperature Sensor Application

3.8.1 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.

3.8.2 Modules

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

tempsensor.c – main application module that has the declaration of attributes, clusters, simple descriptor, initialization and process functions.

tempsensor.h – header file for application module.

3.9 SensorTag Sample Switch/Temperature Sensor Application

3.9.1 Introduction

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

3.9.2 Modules

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

sensortagapp.c – main application module that has the declaration of attributes, clusters, simple descriptor, initialization and process functions.

sensortagapp.h – header file for application module.

3.10 Main Functions

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

`<Sample_App>_task()` – Application task entry point:
- Calls function to register the function pointers to NV drivers.
- Calls application initialization function.
- Calls task process function.

`<Sample_App>_initialization()` – Initializes the Application task:
- Initializes timers.
- Peripheral initialization KEY, LCD and LED.
- Registers application as an ICall dispatcher.
- Calls function to initialize Z-Stack.

`<Sample_App>_initializeStack()` – Initializes the Z-Satck:
- Registers the application's endpoint and its simple descriptor.
- Registers callbacks for Z-Stack indications.
- Register the EZ Mode data.
- Starts the Z-Stack thread.
- Registers the ZCL General Cluster's callbacks.
- Register the Application to receive the unprocessed Foundation command/response messages.
- Registers the application's attribute list.

*<Sample_App>*_process() – This is the "task" of the Sample Application. It handles ICall messages ICALL_SERVICE_CLASS_ZSTACK coming from the Z-Stack Thread and the following events, which may vary depending on the Sample Application:

- DLSAPP_KEY_EVENT: Handle physical key-presses.
- DLSAPP_IDENTIFY_TIMEOUT_EVT: Identify timeout expired.
- DLSAPP_MAIN_SCREEN_EVT: A message will be send to the display.
- DLSAPP_EZMODE_NEXTSTATE_EVT: EZMode event to transition to next state in the EZMode state machine.
- DLSAPP_EZMODE_TIMEOUT_EVT: EZMode timeout expired and event generated.

*<Sample_App>*_processZStackMsgs() – Processes events from Z-Stack:

- zstackmsg_CmdIDs_DEV_STATE_CHANGE_IND: Handle device's NWK state change.
- zstackmsg_CmdIDs_ZDO_MATCH_DESC_RSP: Processes a Match_des_rsp message received over the air.
- zstackmsg_CmdIDs_AF_INCOMING_MSG_IND: Processes incoming ZCL messages.

*<Sample_App>*_processAfIncomingMsgInd () - This function processes AF incoming messages and sends them to the ZCL message processor.

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

*<Sample_App>*_displayLcdUpdate() – Update LCD display according to sample application functionality. Specific LCD interface is described in [3].

*<Sample_App>*_processIdentifyTimeChange() - Handle blinking of the identification LED

*<Sample_App>*_processIdentifyCallback() – This callback will be called by ZCL to initiate Identification. It calls *<Sample_App>*_ processIdentifyTimeChange() to start blinking the respective LED.

*<Sample_App>*_ ezmodeCallback() – 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\SampleDoorLock\CC26xx*.*

   To

   Project\zstack\HomeAutomation\YourApp\ CC26xx*.*

2. Rename the files to match your application's name:

   Under Projects\zstack\HomeAutomation\YourApp\ CC26xx, rename the files as follows:
   - dlsapp.c → yourapp.c

- dlsapp.hc → yourapp.h

Etc.

3. In Projects\zstack\HomeAutomation\YourApp\ CC26xx, rename the following files:
   - SampleDoorLock.ewd→YourApp.ewd
   - SampleDoorLock.ewp→YourApp.ewp
   - SampleDoorLock.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 "SampleDoorLock" (case insensitive) to "YourApp", in YourApp.ewp and YourApp.eww.
5. Using the text editor, replace dlsapp.c with yourapp.c, and dlsapp.h with yourapp.h, in YourApp.ewp.
6. 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:

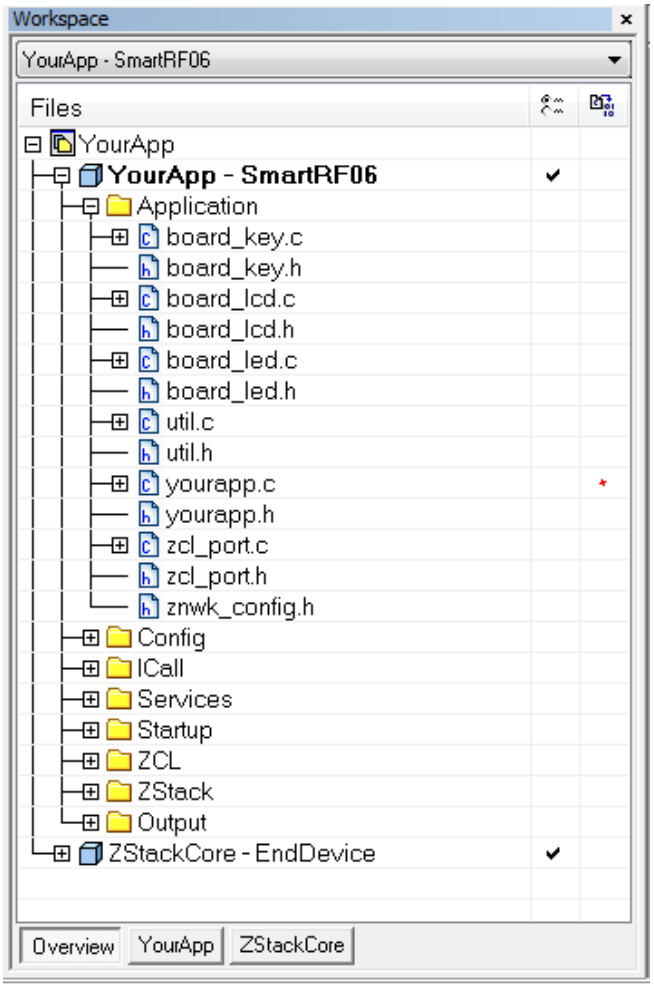

Figure 1 - Application files within YourApp

7. You will have to edit the application files, to replace any occurrence of "dlsapp.h" with "yourapp.h".
8. Now you are ready to manipulate the application code to suite your needs, e.g. modify key-press activities, change the device type, device ID, simple descriptor, modify the supported clusters, attribute list, etc., see section 6.

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 in the compiler's predefined constants section (also called command-line). To disable the option put a small "x" in front of the compiler's predefined constant in the IAR Embedded Workbench preprocessor section of the project (*e.g. xZCL_READ).*

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

- ZCL_STANDALONE

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

- ZCL_READ
- ZCL_WRITE
- ZCL_BASIC
- ZCL_ON_OFF
- ZCL_IDENTIFY
- ZCL_DOORLOCK
- ZCL_EZMODE
- ZCL_GROUPS
- ZCL_SCENES
- ZCL_GEN_MAX_SCENES=5
- ZCL_GEN_SCENE_EXT_LEN=4

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).

SampleDoorLock:

- ZCL_GROUPS
- ZCL_SCENES
- ZCL_DOORLOCK

SampleDoorLockController:

- ZCL_GROUPS
- ZCL_SCENES
- ZCL_DOORLOCK

SampleSwitch:

- ZCL_ON_OFF
- ZCL_HVAC_CLUSTER

SampleTemperatureSensor:

- ZCL_TEMPERATURE_MEASUREMENT

SensorTag Sample Switch/Temperature sensor:

- `ZCL_ON_OFF`
- `ZCL_HVAC_CLUSTER`
- `ZCL_TEMPERATURE_MEASUREMENT`

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 generate a response.

Each cluster may have zero or more attributes. All of the attributes can be found in the <*sampleapp*>*.c* file, where "sampleapp" is replaced with the given sample application (e.g. *dlsapp.c* for the sample doorlock application). Attributes describe the current state of the device, or provide information about the device, such as whether a doorlock is currently locked or unlocked.

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 `zdlAttrs[ ]`, in the <*sampleapp*>*.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.

Read and write attribute functionality is being handled by ZCL. The application needs to control configure reporting and discover functionality.

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 <*sampleapp*>*.c* file. For this DoorLock example, this file is: *dlsapp.c*. Locate the section defined as `zdlAttrs[DLSAPP_MAX_ATTRIBUTES]` 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 *)&zdlNumOfMaxPINLength
```



```
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 `DLSAPP_MAX_ATTRIBUTES` macro in the *dlsapp.c* file. Also within that file, define the external variable using proper coding conventions:

```
static uint8 zdlNumOfMaxPinLength
```

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.

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

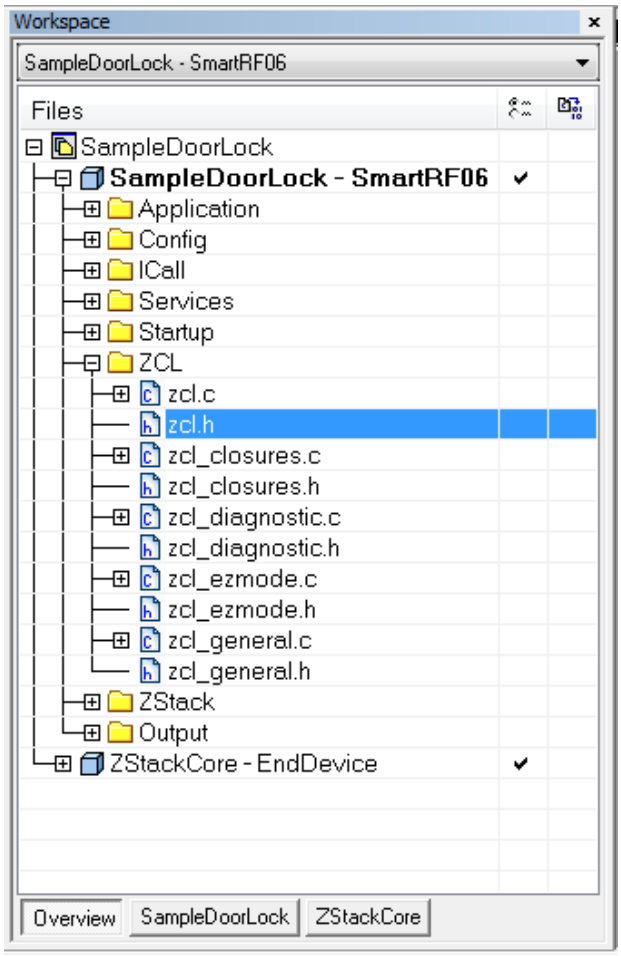

Figure 3 - List of Cluster Source Files in Profile Folder

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. ZCL 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 an Identify command reaches the application layer on a device, the cluster's callback record list points the command to the Identify callback function: `DLSApp_processIdentifyCallback()`.

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, or if the command does not have a specific response and a default response was requested in the original command, ZCL will take care of generate it.

```
/*********************************************************************
 * ZCL General Profile Callback table
 */
static zclGeneral_AppCallbacks_t cmdCallbacks =
{
    NULL,                                   // Basic Cluster Reset command
    DLSApp_processIdentifyCallback,         // Identify command
#ifdef ZCL_EZMODE
    NULL,                               // Identify EZ-Mode Invoke command
    NULL,                               // Identify Update Commission State command
#endif
    NULL,                                   // Identify Trigger Effect command
    DLSApp_identifyQueryResponseCallback, // 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
#ifdef ZCL_LEVEL_CTRL
    NULL,                                 // Level Control Move to Level command
    NULL,                                 // Level Control Move command
    NULL,                                 // Level Control Step command
    NULL,                                 // Level Control Stop command
#endif
#ifdef ZCL_GROUPS
    NULL,                                   // Group Response commands
#endif
#ifdef ZCL_SCENES
    NULL,                                   // Scene Store Request command
    NULL,                                   // Scene Recall Request command
    NULL,                                   // Scene Response command
#endif
#ifdef ZCL_ALARMS
    NULL,                                   // Alarm (Response) commands
#endif
#ifdef SE_UK_EXT
    NULL,                                   // Get Event Log command
    NULL,                                   // Publish Event Log command
#endif
    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.* `DLSApp_handleKeys()`, on switch RIGHT for all Home Automation Sample Applications. Since 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)

The network configuration is stored in NV memory; EZ-Mode has no method for removing bindings directly, other than to factory reset a device. To reset NV memory to factory defaults in SmartRF06 + CC26xx, 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.

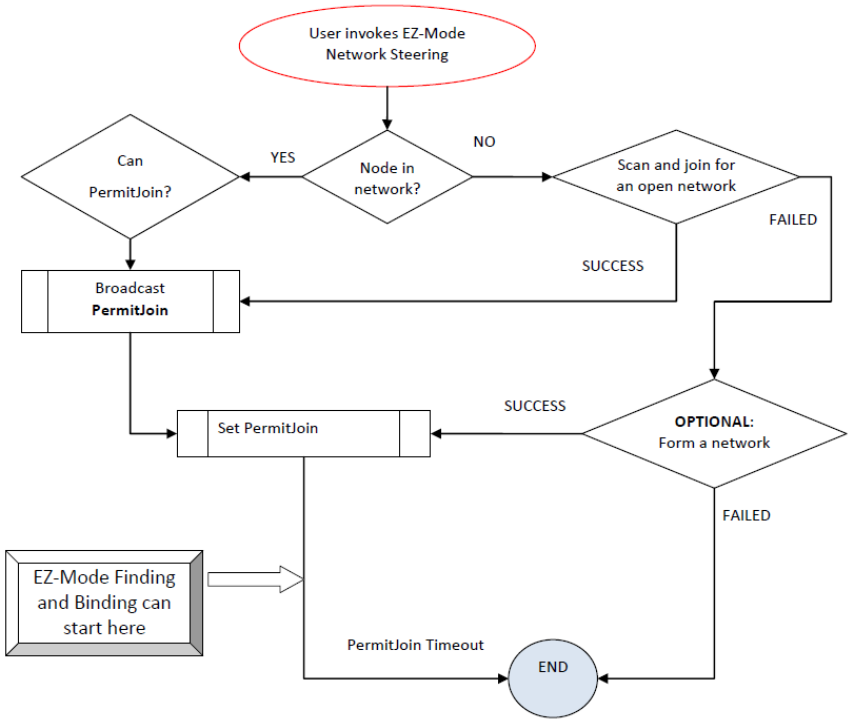

Figure 5 - EZ-Mode Network Steering Flowchart

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.

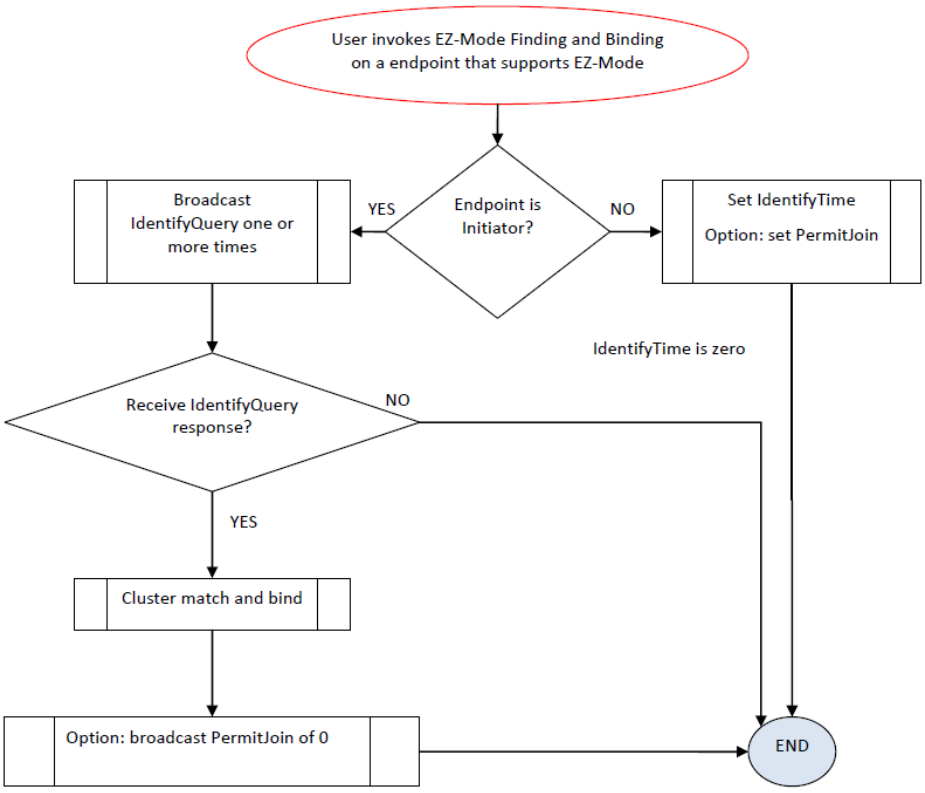

Figure 6 – EZ-Mode Finding and Binding Flowchart

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 SampleDoorLock application informs EZ-Mode to the overall timeout, and when it's time to process the next state.

```
#if defined (ZCL_EZMODE)
            if(events & DLSAPP_EZMODE_NEXTSTATE_EVT)
            {
                // going on to next state
                zcl_EZModeAction(EZMODE_ACTION_PROCESS, NULL);

                // Clear the event, until next time
                events &= ~DLSAPP_EZMODE_NEXTSTATE_EVT;
            }
```



```
            if(events & DLSAPP_EZMODE_TIMEOUT_EVT)
            {
                // EZ-Mode timed out
                zcl_EZModeAction(EZMODE_ACTION_TIMED_OUT, NULL);

                // Clear the event, until next time
                events &= ~DLSAPP_EZMODE_TIMEOUT_EVT;
            }
#endif // ZLC_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`.

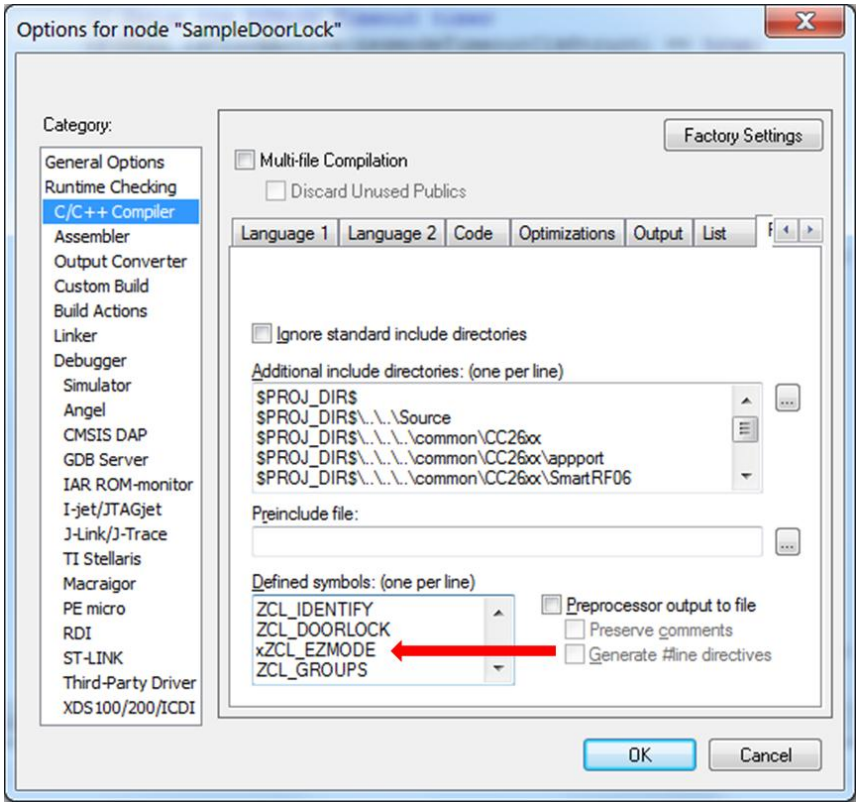

Figure 7 – To Disable EZ-Mode, disable ZCL_EZMODE

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. `xZCL_EZMODE` means EZ Mode 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.

8.1 User Interface

The following user interface elements should be in each application:

- Some way to invoke EZ-Mode on each endpoint
- Some way to cause the application action (e.g. a switch turning on a light)
- Set up battery powered devices as sleepy devices