Simple API for Z-Stack

Document Number: SWRA196

Texas Instruments, Inc.
San Diego, California USA

Version	Description	Date
1.0	Initial Release.	05/22/2007
1.1	Updated for 1.4.3 release.	11/30/2007
1.2	Added MSP2618 platform.	04/19/2008
1.3	Corrected function names for zb_BindDevice, zb_AllowBind, and zb_PermitJoiningRequest.	02/09/2009
1.4	Updated for 2.2.0 release	4/2/2009

Table of Contents

1. Introduction

1.1 Purpose

This document is a tutorial and reference guide for the SimpleApp sample application distributed with the Texas Instruments Z-Stack ™ ZigBee protocol stack (ZigBee Compliant Platform).

The SimpleApp sample application is intended to present a simplified ZigBee API for the application developer.

- Reduced set of API functions and callback events
- Simplified stack startup procedure
- Runtime stack configuration through the use of Z-Tool for Windows

1.2 Scope

This document is limited to developing applications using the Simple API for Texas Instruments Z-Stack ZigBee compliant Protocol Stack. Additional documentation provided with the Z-Stack package such as the Z-Stack Developer's Guide and the Z-Stack API Guide take a more in depth look at application development with Z-Stack's standard programming model.

2. Introduction to ZigBee

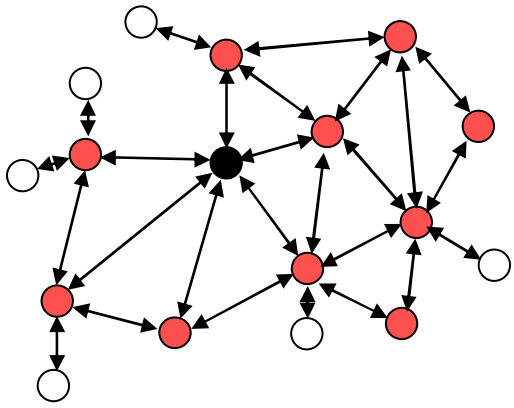
2.1 What is ZigBee

ZigBee is an open and global standard for wirelessly networked control and monitoring solutions that are reliable, cost-effective, low-power. ZigBee utilizes IEEE 802.15.4 compliant radios operating in the 2.4 GHz, 915 MHz, and 868 MHz ISM, Industrial, Scientific and Medical, radio bands. Applications of ZigBee include home and building automation, industrial controls, energy management and automated metering.

A ZigBee network is a self-configuring, multi-hop network with battery-powered devices. This means that two devices that wish to exchange data in a ZigBee network may have to depend on other intermediate devices to be able to successfully do so. Because of this cooperative nature of the network, it is required that each device perform certain networking functions. These functions are determined by the logical ***device type***.

2.2 Device Types

There are three logical device types in a ZigBee network – (i) Coordinator (ii) Router and (iii) End-device. A ZigBee network consists of a Coordinator node and multiple Router and End-device nodes. Note that the device type does not in any way restrict the type of application that may run on the particular device.



An example network is shown in the diagram above, with the ZigBee coordinator (in black), the routers (in red) and the end devices (white).

2.2.1 Coordinator

This is the device that "starts" a ZigBee network. It is the first device on the network. The coordinator node chooses a channel and a network identifier (also called PAN ID) and then starts the network.

The coordinator node can also be used, optionally, to assist in setting up security and application-level bindings in the network.

Note that the role of the Coordinator is mainly related to starting up and configuring the network. Once that is accomplished, the Coordinator behaves like a Router node (or may even go away). The continued operation of the network does not depend on the presence of the Coordinator due to the distributed nature of the ZigBee network.

2.2.2 Router

A Router performs functions for (i) allowing other devices to join the network (ii) multi-hop routing (iii) assisting in communication for its child battery-powered end devices.

In general, routers are expected to be active all the time and thus have to be mains-powered.

A router caches messages destined for its children until a child wakes and requests data. When a child needs to transmit a message, the child sends the data to the parent router. The router then takes responsibility for delivering the message, performing any associated retransmission, and awaits acknowledgement if necessary. This frees the end device to return to sleep.

It is important to note that a router is allowed to be the originator or destination of network traffic. Therefore, a router can play a dual role serving as an end application and as a router. Due to the requirement that routers must be constantly ready to relay data, they are generally mains powered rather then run on batteries. If an application does not call for battery powered devices, it can be advantageous to implement all of the end applications as routers.

2.2.3 End-device

An end-device has no specific responsibility for maintaining the network infrastructure, so it can sleep and wake up as it chooses. End-devices only wake periodically to send and/or receive data to/from their parent. Therefore end devices can be powered by batteries for long periods of time.

2.3 Addressing

ZigBee devices have two types of addresses. A 64-bit *IEEE address* (also called *MAC address* or *Extended address*) and a 16-bit *network address* (also called *logical address* or *short address*).

The 64-bit address is a globally unique address and is assigned to the device for its lifetime. It is usually set by the manufacturer or during installation. These addresses are maintained and allocated by the IEEE. More information on how to acquire a block of these addresses is available at http://standards.ieee.org/regauth/oui/index.shtml

The 16-bit address is assigned to a device when it joins a network and is intended for use while it is on the network. It is only unique within that network. It is used for identifying devices and sending data in the network.

3. Overview of Z-Stack's Simple API

3.1 What is Z-Stack

Z-Stack is Texas Instrument's implementation of the ZigBee specification. It is certified as a ZigBee Compliant Platform (ZCP) by the ZigBee Alliance. It consists of the following components.

- HAL (Hardware abstraction layer)
- OSAL (Operating system abstraction layer)
- ZigBee Stack + IEEE 802.15.4 MAC
- User Application
- MT (Monitor Test) – Used to communicate with a PC-based test tool via the UART.

The following services are provided by the simplified ZigBee API (see API Reference Guide for more details)

- Initialization
    - zb_SystemReset
    - zb_StartRequest
- Configuration
    - zb_ReadConfiguration
    - zb_WriteConfiguration
    - zb_GetDeviceInfo
- Discovery (device, network and service discovery)
    - zb_FindDeviceRequest
    - zb_BindDevice
    - zb_AllowBind
    - zb_PermitJoiningRequest
- Data transfer
    - zb_SendDataRequest
    - zb_ReceiveDataIndication

3.2 How to commission devices into a network

Each device has a set of configuration parameters (see Configuration parameters) that can be configured (for example, by a PC tool or an external microcontroller). The configuration parameters have default values that are defined in code.

Each *"network-specific"* configuration parameter should be set to the same value in all devices that will be part of the network.

The *"device-specific"* configuration parameters can be set to different values for each device. But the **ZCD_NV_LOGICAL_TYPE** must be set so that (i) there is exactly one device configured as a coordinator (ii) all battery powered devices are configured as end-devices.

Once this done, the devices can be powered-up in any order. The coordinator device will start the network and the other devices will find and join it.

The coordinator device will scan all channels specified in the **ZCD_NV_CHANLIST** configuration parameter and pick a channel that has the least energy level. If more than one channel has low energy level, the coordinator will pick the channel with the least number of existing ZigBee networks. The coordinator will choose the network identifier specified in the **ZCD_NV_PANID** parameter.

The routers and end-devices will scan the channels specified in **ZCD_NV_CHANLIST** configuration parameter and try to find the network with the identifier specified in the **ZCD_NV_PANID** parameter.

3.3 How to bind devices

A binding is a logical link between two devices at the application layer. Multiple bindings can be created on a device, one for each type of data packet. In addition, a binding may have more than one destination device (one-to-many bindings).

For example, in a lighting network with multiple switches and lights, each switch will control one or more light. In that case, a binding should be created in each switch. This allows the application to send the data packets without knowing the actual destination address.

Once a binding is created on the source device, the application can send data without specifying a destination address (in the call to zb_SendDataRequest(), the invalid address - 0xFFFE should be used as the destination). This will cause the stack to look up the destination in its internal binding table based on the command identifier of the packet.

There can be more than one destination in the binding entry. In that case, the stack will automatically send a copy of the packet to each destination specified in the binding entry.

Also, if the NV_RESTORE compile option is enabled when building the image, the stack will save the binding entries to non-volatile ram This is useful in the device has an accidental reset (or if the batteries need to be changed on the device), the device can recover automatically without the user having to setup the bindings again.

There are two mechanisms available to configure device bindings. If the extended address of the destination device is known, the zb_BindDevice() can be to create a binding entry.

If the extended address is not known, a “push button” strategy may be employed. In this case, the destination device is first put in a state where it will respond to match requests by issuing the zb_AllowBindResponse(). Then the zb_ BindDevice() is issued on the source device with a null address.

In addition, bindings can be setup by using an external commissioning tool.

Note that bindings can only be created between “complementary” devices. That is, the binding will only succeed if both devices have registered the same command_id in their simple descriptor structures and one device has the command as an “output” while the other device has it as an “input”.

3.4 How to develop a simple private application profile

The following is a way to use the simple api to develop an application.

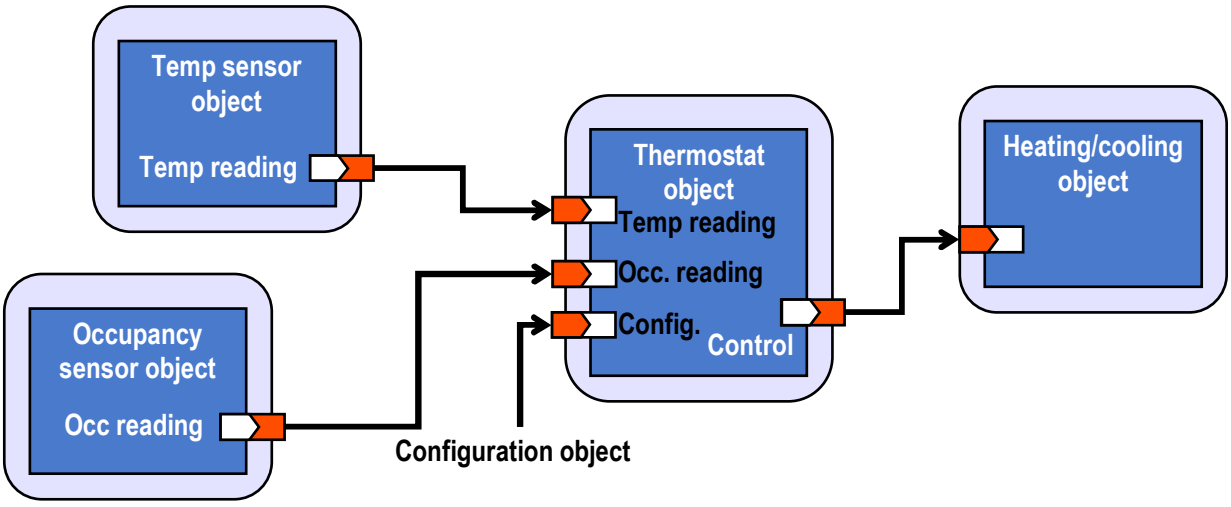
- Identify all the devices in the application
  - e.g. temperature sensor, occupancy sensor, thermostat, heating unit and remote control
  - assign a device_id (unique 16bit identifier) to each of them

- Identify the “*commands*” that need to be exchanged between these devices and assign a unique 16bit command_id to each of them. For example,

  - temperature reading
  - occupancy reading
  - thermostat setting object
  - heating/cooling unit control object

- For each "*command*", identify the devices that "produce" (output) and "consume" (input) it
  - The temperature reading is *"output"* from the temp sensor and *"input"* to the thermostat
  - The occupancy reading command is *"output"* from the occupancy sensor device and *"input"* to the thermostat
  - etc.
  -
- Create the simple descriptor structures for each of the devices. This includes
  - Assigning a device identifier and device version to each of the devices
  - Specifying the list of *"output"* and *"input" command's* for that device.
  - Specify a profile id. This is a 16-bit value that identifies uniquely the application profile. These are assigned by the ZigBee Alliance.

- For each "*command*"
  - define the format of the message being exchanged and its interpretation
  - e.g. temperature value can be exchanged as
    - (format) "*an 8bit value"*
    - (interpretation) *"0 indicates 0oC and 255 indicates 64oC in steps of 0.25o C"*



- Write the device application for each device

  - Device with *"output" commands* should be able to generate the packet either periodically or when an external event occurs.
  - Device with *"input" commands* should handle the reception of these packets and parse the payload.

- Identify a binding strategy so that the devices will be able to exchange packets correctly. See the example sample application on how to do this.

4. Example applications

4.1 Sensor data collection application

A collection of sensor nodes record temperature and battery level readings and send them to a collection node for off-line processing. Typically there will be a single collector node for a network that will receive sensor readings from all devices and either display them or send them offline for further processing. To enhance reliability and to do load-balancing, some networks may have more than one collector node that will accept reports from the sensor devices.

The application must be able to

- Form a network automatically.
- Sensor devices must be able to discover and bind to a collector node automatically upon joining the network.
- Sensor devices must send data periodically to the collector node with end-to-end acknowledgement.
- If sensor device does not receive an acknowledgement from its collector node, it will remove its binding to that collector. It will then rediscover and rebind to (possibly another) collector node.

Devices:

The Sample application project has two device configurations that demonstrate this network - the SimpleSensor device and the SimpleCollector device.

The SimpleSensor device is configured as an end-device since sensors are typically battery powered. The SimpleCollector device is configured as coordinator/router device.

Commands:
There is a single application command – a SENSOR_REPORT_CMD_ID command. This is defined as an "output" for the sensor and as an "input" for the collector device. This command message has a two byte payload. The first byte indicates the type of reading (temperature or battery reading in this case). The second byte indicates the sensor reading level. The temperature value is in degrees centigrade with a range of 0 to 99. The battery level reading is in units of 0.1V and in the range of 0 to 3.75V (on the cc2430 devices) or 3.0V (for the msp430 devices).

Note that the temperature and battery readings will not be very accurate since the parameters have to be calibrated for the hardware. See the source code of the SimpleSensor to adjust the calibration parameters.

Discovery and Binding:

After joining the network, the SimpleSensor device tries to discover and bind itself to a collector. If it discovers more than one collector device, it will pick the first one that responds. If it cannot find a collector node, it will continue searching periodically.

After joining/starting the network, a SimpleCollector device has to be placed in the Allow Bind mode to respond to binding requests from the sensor device. This is achieved in the sample application by pressing SW1 on the device. This will enable the Allow Bind mode on the device and turn on LED1. The device can be removed from this mode by pressing S2 which will disable the Allow Bind mode and turn off LED1.

Packet transmission and reception

After successfully binding, a sensor device will being reading two types of sensor inputs – a temperature reading and a battery level reading and transmit the data in a REPORT command packet to the collector.

The reporting of values is done with the end-to-end acknowledgment turned on. If an ack is not received for any packet, this will be indicated to the application via the zb_SendDataConfirm. Then the sensor device application will remove its existing binding to the collector and try to bind again.

The collector node that receives the sensor packets will send over the serial port. These can be viewed on a PC through HyperTerminal by connecting a serial (or usb-to-serial) cable.

Sample application usage

Program some devices in the SimpleCollector and SampleSensor configurations as described in section 5.3. Make sure that only one of the SimpleCollector is configured as a coordinator while the rest are routers.

After the devices have powered-up and formed a network, place one of the SimpleCollector devices in the Allow Bind mode by pressing SW1. This will turn on LED1 on that device.

The sensor devices will automatically discover and bind to this device. They will begin reporting temperature and battery readings to this device. The LED1 is turned ON when the device is reporting sensor readings to a collector.

On the SimpleCollector device, any received sensor reports are written to the serial port. By connecting a serial (or usb-to-serial) cable to the collector device and opening HyperTerminal, the sensor reports can be viewed on a PC.

Place another SimpleCollector device in the Allow Bind mode and turn off (power-off) the original device that was in this mode. This will cause the sensors to lose communication to their collector node. They will then remove bindings to their collector node and will find the new collector node. By connecting the second collector node to the PC, it can be seen that the sensors will now send their reports to it.

4.2 Preconfigured in-home network

The application consists of a set of light controller devices and light switch. The user must be able to

- Setup these devices into a network automatically.
- Create bindings from each switch to one or more lights.
- Toggle the light by issuing commands from the switch device.
- Reassign bindings for a switch to different lights.
- Add new lights or switches purchased later to the network.

Devices: There are two application device types to demonstrate this sample application – light switch and light controller.

The sample application project has the SimpleSwitch configuration that is configured as an end-device and the SimpleController configuration that is configured as a coordinator/router device.

When the device is first turned on, it comes up in a “HOLD” state with LED1 flashing.

For the light controller device in this state, pressing SW1 will cause the device to startup as a coordinator while pressing SW2 will cause it startup as a router.

For the light switch device in this state, pressing either SW1 or SW2 will cause it come as an end-device.

Commands: There is a single application command – a TOGGLE command. This is defined as an "output" for the switch and as an "input" for the controller. This command message has no other parameters besides the command identifier.

Binding: The "push button" binding is used.

To create a binding between a switch and a controller, the controller is first put in an Allow Bind mode. This is followed by issuing a Bind request on the switch (within the timeout period). This will create a binding from the switch to the controller.

A switch can be bound to more than one controller by repeating the above process.

To reassign the bindings for a switch, the Bind request is issued with a delete parameter. This will remove all bindings for that switch. It can now be bound to other controllers by following above procedure.

Sample application usage

Program some devices in the SimpleController and SimpleSwitch configurations as described in section 5.3. Make sure that only one of the SimpleController is configured as a coordinator while the rest are routers.

After the devices have joined to the network, the controls are used in following manner to creating bindings.

Place a controller in the Allow Bind mode by pressing SW1 on that device. On the light switch, press SW1 (within 10 seconds) to issue the bind request. This will cause it to bind to the controller device that was in the Allow Bind mode. When the switch has successfully created the binding, LED1 is turned ON (or blinking if POWER_SAVING is turned on) for that device.

After that, S2 can be pressed on the switch device to send the TOGGLE command. This will cause LED1 on the corresponding controller device to be toggled.

If S3 is available on the switch device, it can be used to remove all bindings for that device.

4.3 Using the sample applications

The sample applications can be found in the Z-Stack installation folder under:

- Projects\zstack\Samples\SimpleApp\xxxx, where xxxx represents the hardware platform
  - SimpleApp.eww – IAR Embedded Workspace workspace file
  - SimpleApp.ewp– IAR Embedded Workspace project file
- Source
  - Implementation of the sample application.

See the Z-Stack user's guide document corresponding to the appropriate hardware platform for details on building the IAR projects and downloading the code and configuring the Z-Stack.

The sample applications utilize two buttons for user input. The joystick is used to give user input. SW1 is the joystick Up position and SW2 is the joystick Right position.

To give indication to the user, two LED's are used. Upon device power-up, the following LED indications are available.

- LED1 is blinking: On the CC2430 platforms, this happens when the device powers-up and finds that the IEEE address is uninitialized (all F's). The device then waits in a loop for the user to press SW5 (pushing down on the joystick). This will cause the device to pick a random address.

- LED2 is blinking: The device has powered-up but not yet started the ZigBee functionality. It is in an idle state waiting for user input for configuration.

On the end-device's (sensor and switch devices), pressing either SW1 or SW2 will cause the devices to start the ZigBee functionality.

On the coordinator/router devices (controller and collector), pressing SW1 will cause the device to configure itself as a coordinator while SW2 will cause it to configure itself as a router. After the configuration, the device will start the ZigBee functionality.

After successfully starting/joining a ZigBee network, LED3 (on the CC2x30 platforms) or LED2 (on the msp430 platform) will be turned on to give user indication.

5. API Reference Guide

This section consists of the following parts:

- API Functions
- Callback functions
- Configuration Properties

5.1 API Functions

5.1.1 zb_SystemReset

The *zb_SystemReset* function resets the device. The device reads the configuration properties at this time.

The **ZCD_NV_STARTUP_OPTION** configuration property controls the behavior of the device at startup.

If it indicates an erase of the configuration properties (ZCD_STARTOPT_CLEAR_CONFIG is set), the configuration properties will be erased and written with their default values.

If it indicates an automatic ZigBee start (ZCD_STARTOPT_AUTO_START is set), the ZigBee stack will also be started.

Note: The *zb_ SystemReset* function can be called after a call to *zb_WriteConfiguration* to restart the device with the updated configuration.

Note: When *zb_ SystemReset* is called, all volatile memory in the system is reset.

5.1.1.1 Prototype

```
void zb_SystemReset ( void )
```

5.1.1.2 Parameters

None

5.1.1.3 Return Value

None

5.1.2 zb_StartRequest

The *zb_StartRequest* function starts the ZigBee stack. If the startup options indicated that previous network state should be restored (ZCD_STARTOPT_CLEAR_CONFIG is set), then the device will simply load the previously saved network state and being functioning on the ZigBee network.

Otherwise, the device will either start a new network or attempt to join an existing network depending on whether it is configured to be a coordinator or router/end-device. In this case, the device will delay the startup by a value indicated in the configuration property **ZCD_NV_START_DELAY.**

The *zb_StartConfirm* callback function is called at the end of the startup process. After the successful completion of this process, the device is ready to send, receive and route packets in the ZigBee network.

5.1.2.1 Prototype

```
void zb_StartRequest ( void )
```


5.1.2.2 Parameters

None

5.1.2.3 Return Value

None

5.1.3 zb_PermitJoiningRequest

The *zb_PermitJoiningRequest* function is used to control the joining permissions and thus allow or disallow new devices from joining the network.

5.1.3.1 Prototype

```
void zb_PermitJoiningRequest ( uint16 destination, uint8 timeout )
```

5.1.3.2 Parameters

destination

The *destination* parameter indicates the address of the device for which the joining permissions should be set. This is usually the local device address or the special broadcast address that denotes all routers and coordinator (0xFFFC). This way the joining permissions of a single device or the whole network can be controlled.

timeout

The *timeout* parameter indicates the amount of time in seconds for which the joining permissions should be turned on.

If *timeout* is set to 0x00, the device will turn off the joining permissions indefinitely. If it is set to 0xFF, the joining permissions will be turned on indefinitely.

5.1.3.3 Return Value

ZB_SUCCESS or an error parameter

5.1.4 zb_BindDevice

The *zb_BindDevice* function establishes or removes a 'binding' between two devices. Once bound, an application on the source device can send messages to the destination device by referencing the *commandId* for the binding. The bindings are stored in the non-volatile memory and restored upon a reset (unless the startup option explicitly requests otherwise). This way, an accidental reset or temporary power loss will not affect the application.

5.1.4.1 Prototype

```
uint8 zb_BindDevice ( uint8 create, uint16 commandId, uint8 *pDestination )
```

5.1.4.2 Parameters

create

The *create* parameter is TRUE to create a new binding, or FALSE to delete an existing binding.

commandId

The *commandId* identifies message for which this binding should apply. Once a binding is setup, the *commandId* can be used in calls to *zb_SendDataRequest* to send data.

pDestination

When adding a binding, the *pDestination* parameter indicates the 64-bit IEEE address of the device to establish the binding with. If the *pDestination* is NULL, then the device will bind with any other device that is in the Allow Binding Mode. For more information about the Allow Binding Mode, see Section 5.1.5.

The *pDestination* should be set to NULL when deleting a binding.

5.1.4.3 Return Value

None

5.1.5 zb_AllowBind

The *zb_AllowBind* function puts the device into the Allow Binding mode for a given period of time. A peer device can establish a binding to a device in the Allow Binding mode by calling *zb_BindDevice* with a destination address of NULL.

5.1.5.1 Prototype

```
void zb_AllowBind ( uint8 timeout )
```

5.1.5.2 Parameters

timeout

The *timeout* parameter indicates the amount of time in seconds to remain in the Allow Binding Mode.

If *timeout* is set to 0xFF, the device will be in the Allow Bind mode for this *commandId* without any timeout.

If *timeout* is set to 0x00, the device will cancel the Allow Bind mode for this *commandId*.

Otherwise, the device will be in the Allow Bind mode for this *commandId* for the specified time. The maximum *timeout* value is 64 (values larger than that are truncated to 64). Only a single *commandId* may use the Allow Bind mode with the timeout at any time.

5.1.5.3 Return Value

None

5.1.6 zb_SendDataRequest

The *zb_SendDataRequest* function initiates transmission of a data packet to a peer device. The destination of the transmission may be the 16-bit short address of the peer device or an invalid address. In the latter case, the data packet would be sent to device(s) with which bindings were previously established for this particular *commandId*.

The *zb_SendDataRequest* function returns immediately. The status of the send data operation is returned to the Application Task via the *zb_SendDataCnf* callback function.

5.1.6.1 Prototype

```
void zb_SendDataRequest ( uint16 destination, uint16 commandId, uint8 len, uint8
*pData, uint8 handle, uint8 txOptions, uint8 radius )
```

5.1.6.2 Parameters

destination

The *destination* parameter indicates the device in the ZigBee network to transmit the data to. The *destination* parameter can be specified in one of three ways:

- Short Address (value from 0x0000 through 0xFFF8) – The 16-bit short address of the actual destination device.
- A Broadcast address with following valid values
  - 0xFFFF - Broadcast to all devices
  - 0xFFFD - Broadcast only to devices with receiver turned ON
  - 0xFFFC - Broadcast only to coordinator and all routers
- An invalid address (0xFFFE) – In this case, the destination device is not specified by the application. Instead, the stack will read the destination address from a previously established binding for the *commandId*,

commandId

The *commandId* parameter specifies the type of command being issued to the peer device.

dataLength

The *dataLength* parameter contains the number of bytes in the pData buffer.

pData

The pData parameter is a pointer to the data to be transmitted.

handle

The handle parameter contains an identifier for the data transmission. This handle is used in the *zb_SendDataConfirm* callback by the stack to identify the transmission.

txOptions

This is a bit mask of the transmission options.

`ZB_ACK_REQUEST` (0x10)- End-to-end acknowledgement and retransmission should be employed in the transmission of this packet. When using acknowledged transmission, the *zb_SendDataConfirm* callback is delayed until the acknowledgement is received. If the *ack* parameter is FALSE, the *zb_SendDataConfirm* callback is called after the radio transmits the data. The *ack* parameter is ignored if the destination is a broadcast address.

radius

The *radius* parameter indicates the maximum number of hops the data can be relayed in the ZigBee network. Setting the *radius* parameter to 0 indicates a default radius of 15 hops. This can be used to limit the propagation of the data packet in a mesh network.

5.1.6.3 Return Value

None

5.1.7 zb_ReadConfiguration

The *zb_ReadConfiguration* function is used to get a Configuration Property from Nonvolatile memory.

5.1.7.1 Prototype

```
void zb_ReadConfiguration( uint8 configId, uint8 len, void *pValue )
```

5.1.7.2 Parameters

configId

The *configId* parameter indicates the configuration property to read. A list of configuration properties and their identifiers can be found in section 5.3

len

The *len* parameter indicates the size of the *pValue* buffer in bytes.

pValue

The *pValue* parameter is a pointer to a buffer that will contain the configuration property.

5.1.7.3 Return Value

None

5.1.8 zb_WriteConfiguration

The *zb_WriteConfiguration* function is used to write a Configuration Property to nonvolatile memory.

5.1.8.1 Prototype

```
void zb_WriteConfiguration( uint8 configId, uint8 len, void *pValue )
```

5.1.8.2 Parameters

configId

The *configId* parameter indicates the configuration property to write. A list of configuration properties and their identifiers can be found in section 5.3

len

The *len* parameter indicates the size of the *pValue* buffer in bytes.

pValue

The *pValue* parameter is a pointer to a buffer that contains value to write to the configuration property.

5.1.8.3 Return Value

None

5.1.9 zb_GetDeviceInfo

The *zb_GetDeviceInfo* function retrieves a Device Information Property.

5.1.9.1 Prototype

```
void zb_GetDeviceInfo ( uint8 parameter, void *pValue )
```

5.1.9.2 Parameters

parameter

The *parameter* indicates the device information property to read. A list of Device Information Properties an

Property	Identifier	Type	Description
ZB_INFO_DEV_STATE	0x00	8-bit	The current state of the ZigBee device. This can take one of the values in the *devStates_t* enumeration in `ZDApp.h` file.
ZB_INFO_IEEE_ADDR	0x01	64-bit	The 64-bit IEEE address of the device (globally unique).
ZB_INFO_SHORT_ADDR	0x02	16-bit	The 16-bit short address of the device (unique within the network).
ZB_INFO_PARENT_SHORT_ADDR	0x03	16-bit	The 16-bit short address of the parent of this device.
ZB_INFO_ PARENT_IEEE_ADDR	0x04	64-bit	The 64-bit IEEE address of the parent of this device.
ZB_INFO_CHANNEL	0x05	8-bit	The channel on which this device is operating. There are 16 channels in the 2.4GHz band numbered from 11 through 26.
ZB_INFO_PAN_ID	0x06	16-bit	The identifier of the ZigBee network that this device is a part of.
ZB_INFO_EXT_PAN_ID	0x07	64-bit	The 64-bit IEEE address of the ZigBee coordinator device for this network.

pValue

The *pValue* parameter is a pointer to a buffer that will contain the Device Information Property. Note that the application must ensure that the buffer has sufficient size to allow the property to be copied over.

5.1.9.3 Return Value

None

5.1.10 zb_FindDeviceRequest

The *zb_FindDeviceRequest* function is used to determine the short address for a device in the network. The device initiating a call to zb_FindDeviceRequest and the device being discovered must both be a member of the same network. When the search is complete, the zv_FindDeviceConfirm callback function is called.

5.1.10.1 Prototype

```
void zb_FindDeviceRequest( uint8 searchType, uint8 *searchKey )
```

5.1.10.2 Parameters

searchType

The *searchType* parameter indicates the method of search. The following search types can be used:

- ZB_IEEE_SEARCH

searchKey

The *searchKey* parameter contains information unique to the device being discovered. The *searchKey* parameter is dependant on the *searchType*. The content of the *searchKey* for each *searchType* follows:

- ZB_IEEE_SEARCH – The searchKey is the 64-bit IEEE address of the device being discovered.

5.1.10.3 Return Value

None

5.2 Callback Functions

This section is a reference for the Simple API callback functions implemented by a ZigBee application and called by the ZigBee stack.

5.2.1 zb_StartConfirm

The *zb_StartConfirm* callback is called by the ZigBee stack after a start request operation completes. The Start Confirm Callback notifies the Simple Application Task of the status of the start operation.

If the status is `ZB_SUCCESS`, the device has successfully started or joined the ZigBee network depending on whether it is programmed as a coordinator or router/end-device.

5.2.1.1 Prototype

```
void zb_StartConfirm( uint8 status )
```

5.2.1.2 Parameters

status

5.2.2 zb_BindConfirm

The zb_BindConfirm callback is called by the ZigBee stack after a bind operation completes. The bind confirm callback contains the status of the bind operation.

5.2.2.1 Prototype

```
void zb_BindConfirm( uint16 commandId, uint8 status )
```

5.2.2.2 Parameters

commandId

The *commandId* parameter identifies the binding. This parameter matches the *commandId* passed into the *zb_BindDevice* function to which this callback is in confirmation of.

status

The *status* parameter contains the result of the bind operation.

5.2.3 zb_AllowBindConfirm

The zb_AllowBindConfirm callback is called by the ZigBee stack if a device is in the Allow Bind mode and it responded to a bind request from another device.

5.2.3.1 Prototype

```
void zb_AllowBindConfirm( uint16 source )
```

5.2.3.2 Parameters

source

The *source* parameter contains the address of the source device that requested a binding with this device.

5.2.4 zb_SendDataConfirm

The zb_SendDataConfirm callback function is called by the ZigBee when a send data operation completes

.

When sending data with acknowledgment enabled, the Send Data Confirm Callback is not returned until acknowledgement is received, or a timeout occurs. The latency between zb_SendDataRequest and zb_SendDataConfirm with acknowledgement enabled varies depending on network conditions and the number of hops to deliver the message.

5.2.4.1 Prototype

```
void zb_SendDataConfirm( uint8 handle, uint8 status )
```

5.2.4.2 Parameters

handle

The *handle* parameter identifies the send data operation. The value of the handle matches the value of the handle passed into the *zb_SendDataRequest* function that this callback is in confirmation of.

status

The *status* parameter contains the status of the Send Data operation.

5.2.5 zb_ReceiveDataIndication

The *zb_ReceiveDataIndication* callback function is called asynchronously by the ZigBee stack to notify the application when data is received from a peer device.

5.2.5.1 Prototype

```
void zb_ReceiveDataIndication(uint16 source, uint16 command, uint8 len, uint8 *pData)
```

5.2.5.2 Parameters

source

The *source* parameter contains the 16-bit short address of the device that transmitted the data.

commandId

The commandId parameter contains the command identifier of the binding that the transfer originated from.

len

The *len* parameter contains the size of the pData buffer in bytes.

pData

The *pData* parameter points to the received data.

5.2.6 zb_FindDeviceConfirm

The *zb_FindDeviceConfirm* callback function is called by the ZigBee stack when a find device operation completes.

5.2.6.1 Prototype

```
void zb_FindDeviceConfirm( uint8 searchType, uint8 *searchKey, uint8 *result )
```

5.2.6.2 Parameters

searchType

The search type that was requested for this search operation.

searchKey

The *searchKey* parameter contains information unique to the device being discovered. The *searchKey* parameter is dependant on the *searchType*. The content of the *searchKey* for each *searchType* follows:

- ZB_IEEE_SEARCH – The searchKey is the 64-bit IEEE address of the device being discovered.

result

A pointer to data containing the result of the search. If the search type was ZB_IEEE_SEARCH, then this is a 16-bit address of the device that matched the search.

5.2.7 zb_HandleKeys

The *zb_HandleKeys* function is called by the operating system when a key event is set. This happens if a key press happens on the development board.

5.2.7.1 Prototype

```
void zb_HandleKeys( uint8 shift, uint8 keys )
```

5.2.7.2 Parameters

shift

True if the shift key is pressed down while the key has been pressed. The shift key is only available some hardware development boards.

keys

This indicates the key that has been pressed.

5.2.8 zb_HandleOsalEvent

The *zb_HandleOsalEvent* function is called by the operating system when a task event is set. An application can set a task event using the *osal_set_event* or the *osal_start_timer* functions. For more information about OSAL functions, see the "Z-Stack OS abstraction layer (OSAL) API" document.

5.2.8.1 Prototype

```
void zb_HandleOsalEvent( uint16 event )
```

5.2.8.2 Parameters

event

The *event* parameter contains a bitmask. Each bit in the bitmask corresponds to a task event.

5.3 Configuration parameters

The following list of configuration properties can be written and read from nonvolatile memory using the *zb_WriteConfiguration* and *zb_ReadConfiguration* functions.

Each of the configuration parameters have "default" values that are defined in the code. Once an image is downloaded onto the device, the configuration parameters are initialized to these values.

After a device is programmed with an image, these parameters can be changed by the application or by an external PC tool or an external micro controller. Any changes to the parameters will not take effect unless the device is reset and restarted.

It is possible to erase all the configuration settings and restore them to the initial default settings by setting the startup option parameter appropriately.

The configuration parameters are divided into “network-specific” and “device-specific” parameters. The “network-specific” configuration parameters should be set to the same value for all devices in a network. The “device-specific” parameters can be set to different values on each device.

5.3.1 Network specific parameters

ZCD_NV_PANID

Configuration ID: 0x0083; Size: 2bytes; Default value: ZDAPP_CONFIG_PAN_ID in f8wConfig.cfg file.

This parameter identifies the ZigBee network. This should be set to a value between 0 and 0xFFFE. Networks that exist in the same vicinity must have different values for this parameter. It can be set to a special value of 0xFFFF to indicate *“don’t care”*.

ZCD_NV_CHANLIST

Configuration ID: 0x0084; Size: 4bytes; Default value: DEFAULT_CHANLIST in f8wConfig.cfg file.

This parameter is a bit mask of the channels on which this network can operate. Multiple networks that exist in the same vicinity are encouraged to have different values.

ZCD_NV_PRECFGKEY

Configuration ID: 0x0062; Size: 16bytes; Default value: defaultKey[] in nwk_globals.c file.

The 128-bit key that is used for packet security if that functionality is enabled.

ZCD_NV_PRECFGKEYS_ENABLE

Configuration ID: 0x0063; Size: 1byte; Default value: zgPreConfigKeys in ZGlobals.c file.

If security functionality is enabled, there are two options to distribute the security key to all devices in the network.

If this parameter is true, the security keys must be pre-configured on all devices in the network.

If it is set to false, then the key only needs to be configured on the coordinator device. In this case, the key is distributed to each device upon joining by the coordinator. This key distribution will happen in the “clear” on the last hop of the packet transmission and constitutes a brief “period of vulnerability” when a malicious device can capture the key. Hence it is not recommended unless it can be ensured that there are no malicious devices in the vicinity at the time of network formation.

ZCD_NV_SECURITY_LEVEL

Configuration ID: 0x0061; Size: 1byte; Default value: SECURITY_LEVEL in nwk_globals.h file.

The amount of security applied to each packet if the functionality is enabled. It takes values from 1 through 7.

In levels 1 through 3, the packets are not encrypted but they are authenticated with the authentication code of 4, 8 or 16 bytes for each packet.

In level 4, the packet is encrypted but not authenticated. This setting is not recommended.

In levels 5 through 7, the packets are encrypted, In addition, they are authenticated with codes of length 4, 8 and 16 bytes for each packet.

ZCD_NV_BCAST_RETRIES

Configuration ID: 0x002E; Size: 1byte; Default value: MAX_BCAST_RETRIES in ZGlobals.h file.

The maximum number of retransmissions that a device will attempt when transmitting a broadcast packet. The typical range is from 1 through 3.

ZCD_NV_PASSIVE_ACK_TIMEOUT

Configuration ID: 0x002F; Size: 1byte; Default value: PASSIVE_ACK_TIMEOUT in ZGlobals.h file.

The amount of time (in units of 100milliseconds) a device will wait to hear from it neighbor nodes before retransmitting a broadcast packet.

ZCD_NV_BCAST_DELIVERY_TIME

Configuration ID: 0x0030; Size: 1byte; Default value: BCAST_DELIVERY_TIME in ZGlobals.h file.

The amount of time (in units of 100ms) that it takes for a broadcast packet to propagate through the entire network.

Note: This parameter must be set with caution. It must be set to a value of atleast

(**ZCD_NV_BCAST_RETRIES** + 1) * **ZCD_NV_PASSIVE_ACK_TIMEOUT**

To be safe, the actual value should be higher than the above minimum by about 500ms or more.

ZCD_NV_ROUTE_EXPIRY_TIME

Configuration ID: 0x002C; Size: 1byte; Default value: ROUTE_EXPIRY_TIME in f8wConfig.cfg file.

The amount of time (in seconds) for which a route must be idle (*i.e.* no packets are transmitted on that route) before the route entry is marked as expired. An expired entry is not deleted unless that space for a new routing entry.

This can be set to a special value of 0 to turn off route expiry. In this case, route entries are not expired.

5.3.2 Device specific parameters

5.3.2.1 Startup parameters

ZCD_NV_STARTUP_OPTION

Configuration ID: 0x0003; Size: 1byte; Default value: 0

This parameter controls the device startup logic. This is a bit mask of the following values

- ZCD_STARTOPT_CLEAR_CONFIG (0x01) – If this option is set, the device will overwrite all its configuration parameters (except this one) with the "default" settings that it is programmed with. This is used to erase the existing configuration and bring the device into a known state.

  Note: Whe the configuration parameters are overwritten, the *ZCD_NV_STARTUP_OPTION* itself is not overwritten except for clearing the ZCD_STARTOPT_CLEAR_CONFIG bit.

- ZCD_STARTOPT_CLEAR_STATE (0x02) – If this option is set, the device will attempt to clear its network state prior to the reset. This is used if the device was already part of the network and had saved its previous network state.

  Note: The *NV_RESTORE* compile flag must be turned on to use this feature. In that case, this option will be automatically cleared by the stack after the device joins a network. This is so that an accidental reset of the device does not prevent loss of network state. The application has to explicitly set this option before issuing a reset in order to erase the network state.

- ZCD_STARTOPT_AUTO_START (0x04) – If this option is set, the device will start the ZigBee network functions immediately upon powerup. Otherwise, the device will wait until the application explicitly requests a startup.

ZCD_NV_START_DELAY

Configuration ID: 0x0004; Size: 1byte; Default value: START_DELAY in ZGlobals.c file

The minimum delay (in milliseconds) after the zb_StartRequest() is called (or after power-up if the auto-start bit is set in the startup options configuration parameter) before the ZigBee functions are started.

ZCD_NV_EXTADDR

Configuration ID: 0x0004; Size: 8bytes; Default value: Invalid (All F's)

The 64bit extended address of the device.

ZCD_NV_LOGICAL_TYPE

Configuration ID: 0x0087; Size: 1byte; Default value: DEVICE_LOGICAL_TYPE in ZGlobals.h file

The logical type of the device in the ZigBee network. This can be set to one of the following values ZG_DEVICETYPE_COORDINATOR (0x00), ZG_DEVICETYPE_ROUTER (0x01) or ZG_DEVICETYPE_ENDDEVICE (0x02).

If the end-device project is used to build the image, the type will be automatically selected. Otherwise, the type can be configured by the application to either coordinator or router.

5.3.2.2 Poll parameters

(These are only applicable to a battery powered end-device.)

ZCD_NV_POLL_RATE

Configuration ID: 0x0024; Size: 1byte; Default value: POLL_RATE in f8wConfig.cfg

If set to a non-zero value, an end-device will wake up periodically with this duration to check for data with their parent device. The value is specified in milliseconds and can range from 1 to 65000.

If set to zero, the device will not automatically wake up to check for data. Instead, an external trigger or an internal event (set, for example, via the OSAL timer or event interface) can be used to wake up the device.

ZCD_NV_QUEUED_POLL_RATE

Configuration ID: 0x0025; Size: 1byte; Default value: QUEUED_POLL_RATE in f8wConfig.cfg

When an end-device checks for data with its parent and finds that it does have data, it can poll again with a shorter duration in case there is more data queued for it at its parent device.

This feature can be turned off by setting the value to zero.

ZCD_NV_RESPONSE_POLL_RATE

Configuration ID: 0x0026; Size: 1byte; Default value: in RESPONSE_POLL_RATE in f8wConfig.cfg

When an end-device sends a data packet, it can poll again with a shorter duration if the application is expecting to receive a packet in response.

This feature can be turned off by setting the value to zero.

Note: The setting of the queued and response poll rates has to be done with caution if the device is sending and receiving at the same time or if the device is sending data too fast.

If the device is sending data too fast, setting a queued poll rate with a higher duration than the sending rate will cause the poll event to be continuously rescheduled to the future. Then the device will never poll for data with its parent and consequently it may miss any packets destined for it.

ZCD_NV_POLL_FAILURE_RETRIES

Configuration ID: 0x0029; Size:1byte; Default value: MAX_POLL_FAILURE_RETRIES in f8wConfig.cfg file.

The number of times an end-device will fail contacting its parent before invoking mechanism to find a new parent.

ZCD_NV_INDIRECT_MSG_TIMEOUT

ConfigurationID: 0x002B; Size:1byte; Default value: NWK_INDIRECT_MSG_TIMEOUT in f8wConfig.cfg file.

The amount of time (in seconds) that a router or coordinator device will buffer data meant for its end-device children. It is recommended that this is atleast greater than the poll rate to ensure that end-device will have a chance to wakeup and poll for the data.

5.3.2.3 End-to-end acknowledgement parameters

End-to-end acknowledgements and retransmissions are only applicable if the application explicitly requested it when sending a data packet by setting the appropriate bit in the txOptions parameter in the zb_SendDataRequest() call.

ZCD_NV_APS_FRAME_RETRIES

Configuration ID: 0x0043; Size: 1bytes; Default value: APSC_MAX_FRAME_RETRIES in f8wConfig.cfg

The number of restransmissions performed on a data packet at the application layer if the packet was transmitted with the end-to-end ack option enabled.

ZCD_NV_APS_ACK_WAIT_DURATION

Configuration ID: 0x0044; Size: 2bytes; Default value: APSC_ACK_WAIT_DURATION_POLLED in f8wConfig.cfg file.

The amount of time (in milliseconds) a device will wait after transmitting a packet with end-to-end acknowledgement option set for the acknowledgement packet from the destination device. If the acknowledgement packet is not received by this time, the sending device will assume failure and attempting another retransmission.

Note: This must be set with caution if the destination (or source) device is an end-device, since those devices will not wake up often and hence will add an additional delay of the packet (data or acknowledgement packet) beyond what is caused normally by the network.

5.3.2.4 Miscellaneous

ZCD_NV_BINDING_TIME

Configuration ID: 0x0046; Size: 2bytes; Default value: APS_DEFAULT_MAXBINDING_TIME in ZGlobals.h

The amount of time (in milliseconds) a device will wait for a response to a binding request.

ZCD_NV_USERDESC

Configuration ID: 0x0081; Size: 17bytes; Default value: zero

An optional 16bytes (plus 1 byte of overhead) of user-defined data that can be configured in a device so that it can easily identified or described later.

