TEXAS INSTRUMENTS

Z-Stack TI-RTOS Developer's Guide

Texas Instruments, Inc.
San Diego, California USA

Revision	Description	Date
1.0	Initial release	01/19/2015

TABLE OF CONTENTS

1. INTRODUCTION 1
1.1 PURPOSE 1
1.2 SCOPE 1
1.3 ACRONYMS 1
1.4 REFERENCE DOCUMENTS 1
1.5 NOTES 2

2. ZIGBEE 2
2.1 DEVICE TYPES 2
2.1.1 Coordinator 2
2.1.2 Router 3
2.1.3 End-device 3
2.2 STACK PROFILE 3

3. ADDRESSING 4
3.1 ADDRESS TYPES 4
3.2 NETWORK ADDRESS ASSIGNMENT 4
3.2.1 Stochastic Addressing 4
3.3 ADDRESSING IN Z-STACK 4
3.3.1 Unicast 5
3.3.2 Indirect 5
3.3.3 Broadcast 6
3.3.4 Group Addressing 7
3.4 IMPORTANT DEVICE ADDRESSES 7

4. BINDING 8
4.1 BUILDING A BINDING TABLE 8
4.1.1 ZigBee Device Object Bind Request 8
4.1.1.1 The Commissioning Application 8
4.1.1.2 ZigBee Device Object End Device Bind Request 8
4.1.2 Device Application Binding Manager 9
4.2 CONFIGURING SOURCE BINDING 9

5. ROUTING 10
5.1 OVERVIEW 10
5.2 ROUTING PROTOCOL 10
5.2.1 Route Discovery and Selection 11
5.2.2 Route maintenance 11
5.2.3 Route expiry 11
5.3 TABLE STORAGE 11
5.3.1 Routing table 11
5.3.2 Route discovery table 12
5.4 MANY-TO-ONE ROUTING PROTOCOL 12
5.4.1 Many-to-One Routing Overview 12
5.4.2 Many-to-One Route Discovery 12
5.4.3 Route Record Command 13
5.4.4 Many-to-One Route Maintenance 14
5.5 ROUTING SETTINGS QUICK REFERENCE 14

6. ZDO MESSAGE REQUESTS 16

7. PORTABLE DEVICES 18

8. END-TO-END ACKNOWLEDGEMENTS 19

9. MISCELLANEOUS 20
9.1 CONFIGURING CHANNEL 20
9.2 CONFIGURING THE PAN ID AND NETWORK TO JOIN 20
9.3 LEAVE NETWORK 20
9.4 DESCRIPTORS 20
9.5 NON-VOLATILE MEMORY ITEMS 20
9.5.1 Global Configuration Non-Volatile Memory 20
9.5.2 Network Layer Non-Volatile Memory 20

9.6 ASYNCHRONOUS LINKS 21
9.7 MULTICAST MESSAGES 21
9.8 FRAGMENTATION 22
9.9 EXTENDED PAN IDS 22
9.10 REJOINING WITH PRE-COMMISSIONED NETWORK ADDRESS 22

10. SECURITY 23
10.1 OVERVIEW 23
10.2 CONFIGURATION 23
10.3 NETWORK ACCESS CONTROL 23
10.4 KEY UPDATES 23
10.5 TRUST CENTER LINK KEY 24
10.6 JOINING A NETWORK WITH TCLK 24
10.6.1 Multi-hop 24
10.6.2 Single-hop 25
10.7 SECURITY KEY DATA MANAGEMENT 27
10.8 BACKWARDS INTEROPERABILITY 28
10.9 QUICK REFERENCE 28

11. ZMAC LQI ADJUSTMENT 29
11.1 OVERVIEW 29
11.2 LQI ADJUSTMENT MODES 29
11.3 USING LQI ADJUSTMENT 29

12. COMPILE OPTIONS 30
12.1 OVERVIEW 30
12.2 REQUIREMENTS 30
12.2.1 Target Development System Requirements 30
12.3 USING COMPILE OPTIONS 31
12.3.1 Locating Compile Options 31
12.3.1.1 Compile Options In Linker Control Files 31
12.3.1.2 Compile Options in IAR Project Files 34
12.3.2 Using Compile Options 35
12.4 SUPPORTED COMPILE OPTIONS AND DEFINITIONS 36
12.4.1 General Compile Options for the Z-Stack Thread poject 36
12.4.2 ZigBee Device Object (ZDO) Compile Options 38

1. Introduction

1.1 Purpose
This document explains some of the components of the Texas Instruments ZigBee stack and their functioning. It explains the configurable parameters in the ZigBee stack and how they may be changed by the application developer to suit the application requirements.

1.2 Scope
This document describes concepts and settings for the Texas Instruments Z-Stack™ Release. This is a ZigBee-2012 compliant stack for the ZigBee and ZigBee PRO stack profiles.

1.3 Acronyms

AF	Application Framework
AES	Advanced Encryption Standard
AIB	APS Information Base
API	Application Programming Interface
APS	Application Support Sub-Layer
APSDE	APS Date Entity
APSME	APS Management Entity
ASDU	APS Service Datagram Unit
BSP	Board Support Package – taken together, HAL & OSAL comprise a rudimentary operating system commonly referred to as a BSP
CCM*	Enhanced counter with CBC-MAC mode of operation
EPID	Extended PAN ID
HAL	Hardware (H/W) Abstraction Layer
MSG	Message
NHLE	Next Higher Layer Entity
NIB	Network Information Base
NWK	Network
OSAL	Operating System (OS) Abstraction Layer
OTA	Over-The-Air
PAN	Personal Area Network
SE	Smart Energy
ZDO	ZigBee Device Object

1.4 Reference Documents
[1] ZigBee Specification, R20, ZigBee Alliance document number 053474r20.
[2] Z-Stack TI-RTOS API

1.5 Notes

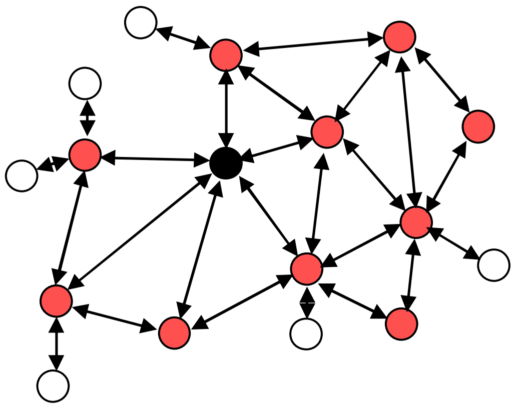
- For CC2630/CC2650 projects, the workspace is split into 2 images, an application image and a Z-Stack image, each image has its own project file. The compile flags and settings in f8wConfig.cfg, `ZGlobals.h` and `nwk_globals.h`, described in this document, used to change Z-Stack settings and behavior, are changed in the Z-Stack thread project.
- The application has control over the default channel list, PAN ID, extended PAN ID and end device poll rate settings with defines in `znwk_config.h` (in the application image project), these settings overwrite the f8wconfig.cfg settings when the application startups through APIs (zstackapi.h).
- In Z-Stack, the device type is usually determined at compile-time via compile options (`ZDO_COORDINATOR` and `RTR_NWK`). All sample applications are provided with separate project files to build each device type.

2. ZigBee

A ZigBee network is a 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, proper functioning requires that each device (i) perform specific networking functions and (ii) configure certain parameters to specific values. The set of networking functions that a device performs determines the role of the device in the network and is called a ***device type***. The set of parameters that need to be configured to specific values, along with those values, is called a ***stack profile***.

2.1 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 (black), the Routers (red), and the End Devices (white).

2.1.1 Coordinator

This is the device that "starts" a ZigBee network. It is the first device on the network. The coordinator node scans the RF environment for existing networks, 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.

2.1.2 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.

2.1.3 End-device

An end device has no specific responsibility for maintaining the network infrastructure, so it can sleep and wake up as it chooses. Thus it can be a battery-powered node.

An end device, with no responsibility for network functionality, can also not sleep and leave its receiver on all the time, but this device will not conserve battery power (needs the compile flag `RFD_RCVC_ALWAYS_ON` set in the Z-Stack Thread project).

Generally, the memory requirements (especially RAM requirements) are lower for an end device.

2.2 Stack Profile

The set of stack parameters that need to be configured to specific values, along with the above device type values, is called a ***stack profile***. The parameters that comprise the stack profile are defined by the ZigBee Alliance.

All devices in a network must conform to the same stack profile (i.e., all devices must have the stack profile parameters configured to the same values).

The ZigBee Alliance has defined two different stack profiles for the ZigBee-2007 specification, Zigbee and Zigbee PRO, with the goal of promoting interoperability. All devices that conform to this stack profile will be able to work in a network with devices from other vendors that also conform to it.

If application developers choose to change the settings for any of these parameters, they can do so with the caveat that those devices will no longer be able to interoperate with devices from other vendors that choose to follow the ZigBee specified stack profile. Thus, developers of “closed networks” may choose to change the settings of the stack profile variables. These stack profiles are called “network-specific” stack profile.

The stack profile identifier that a device conforms to is present in the beacon transmitted by that device. This enables a device to determine the stack profile of a network before joining to it. The “network-specific” stack profile has an ID of 0 while the ZigBee stack profile has ID of 1, and a ZigBee PRO stack profile has ID of 2. The stack profile is configured by the `STACK_PROFILE_ID` parameter in `nwk_globals.h` file.

Normally, a device of 1 profile (ex. ZigBee PRO) joins a network with the same profile. If a router of 1 profile (ex. ZigBee PRO) joins a network with a different profile (ex. ZigBee-2007), it will join as a non-sleeping end device. An end device of 1 profile (ex. ZigBee PRO) will always be an end device in a network with a different profile.

3. Addressing

3.1 Address types

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 within the network.

3.2 Network address assignment

3.2.1 Stochastic Addressing

ZigBee PRO uses a stochastic (random) addressing scheme for assigning the network addresses. This addressing scheme randomly assigns short addresses to new devices, and then uses the rest of the devices in the network to ensure that there are no duplicate addresses. When a device joins, it receives its randomly generated address from its parent. The new network node then generates a “Device Announce” (which contains its new short address and its extended address) to the rest of the network. If there is another device with the same short address, a node (router) in the network will send out a broadcast “Network Status – Address Conflict” to the entire network and all devices with the conflicting short address will change its short address. When the conflicted devices change their address they issue their own “Device Announce” to check their new address for conflicts within the network.

End devices do not participate in the “Address Conflict”. Their parents do that for them. If an “Address Conflict” occurs for an end device, its parent will issue the end device a “Rejoin Response” message to change the end device’s short address and the end device issues a “Device Announce” to check their new address for conflicts within the network.

When a “Device Announce” is received, the association and binding tables are updated with the new short address, routing table information is not updated (new routes must be established). If a parent determines that the “Device Announce” pertains to one of its end device children, but it didn’t come directly from the child, the parent will assume that the child moved to another parent.

3.3 Addressing in Z-Stack

In order to send data to a device on the ZigBee network, the application generally uses the `Zstackapi_AfDataReq ()` function [defined in zstackapi.h]. The destination device to which the packet is to be sent is of type `zstack_AFAddr_t` (defined in `zstack.h`).

```
/**
 * This is a structure used to define an application address.
 * Depending on addrMode, only one shortAddr or extAddr should be used.
 */
typedef struct _zstack_afaddr_t
{
    /** Address Mode */
    zstack_AFAddrMode addrMode;
    /** Address union of 16 bit short address and 64 bit IEEE address */
    union
    {
        /** 16 bit network address */
        uint16_t shortAddr;
        /** 64 bit IEEE address */
        zstack_LongAddr_t extAddr;
    } addr;
    /** Endpoint address element, optional if addressing to the endpoint,
     * can be 0xFF to address all endpoints in a device.
     */
    uint8_t endpoint;
    /** PAN ID - for use with Inter-PAN */
    uint16_t panID;
} zstack_AFAddr_t;
```

Note that in addition to the network address, the address mode parameter also needs to be specified. The destination address mode can take one of the following values (AF address modes are defined in `zstack.h`)

```
/** Address types */
typedef enum
{
    //! Address not present
    zstack_AFAddrMode_NONE = 0,
    //! Group Address (uint16_t)
    zstack_AFAddrMode_GROUP = 1,
    //! Short Address (uint16_t)
    zstack_AFAddrMode_SHORT = 2,
    //! Extended Address (8 bytes/64 bits)
    zstack_AFAddrMode_EXT = 3,
    //! Broadcast Address (uint16_t)
    zstack_AFAddrMode_BROADCAST = 15,
} zstack_AFAddrMode;
```

The address mode parameter is necessary because, in ZigBee, packets can be unicast, multicast or broadcast. A unicast packet is sent to a single device, a multicast packet is destined to a group of devices and a broadcast packet is generally sent to all devices in the network. This is explained in more detail below.

3.3.1 Unicast

This is the normal addressing mode and is used to send a packet to a single device whose network address is known. The `addrMode` is set to `zstack_AFAddrMode_SHORT` and the destination network address is carried in the packet.

3.3.2 Indirect

This is when the application is not aware of the final destination of the packet. The mode is set to `zstack_AFAddrMode_NONE` and the destination address is not specified. Instead, the destination is looked up from a “binding table” that resides in the stack of the sending device. This feature is called Source binding (see later section for details on binding).

When the packet is sent down to the stack, the destination address and end point is looked up from the binding table and used. The packet is then treated as a regular unicast packet. If more than one destination device is found in the binding table, a copy of the packet is sent to each of them. If no binding entry is found, the packet will not be sent.

3.3.3 Broadcast

This address mode is used when the application wants to send a packet to all devices in the network. The address mode is set to `zstack_AFAddrMode_BROADCAST` and the destination address can be set to one of the following broadcast addresses:
`0xFFFF` – the message will be sent to all devices in the network (includes sleeping devices). For sleeping devices, the message is held at its parent until the sleeping device polls for it or the message is timed out (`NWK_INDIRECT_MSG_TIMEOUT` in `f8wConfig.cfg`).
`0xFFFD` – the message will be sent to all devices that have the receiver on when idle (`RXONWHENIDLE`). That is, all devices except sleeping devices.
`0xFFFC` – the message is sent to all routers (including the coordinator).

3.3.4 Group Addressing

This address mode is used when the application wants to send a packet to a group of devices. The address mode is set to `zstack_AFAddrMode_GROUP` and the `addr.shortAddr` is set to the group identifier.

Before using this feature, groups must be defined in the network, see `Zstackapi_ApsAddGroupReq ()` in the Z-Stack TI-RTOS API [2] document.

Note that groups can also be used in conjunction with indirect addressing. The destination address found in the binding table can be either a unicast or a group address. Also note that broadcast addressing is simply a special case of group addressing where the groups are setup ahead of time.

Sample code for a device to add itself to a group with identifier 1:

```
zstack_apsAddGroup_t group;

// Assign yourself to group 1
group.groupID = 0x0001;
group.n_name = 6;  // Group name length
group.pName = “Group1”;
Zstackapi_ApsAddGroupReq ( appEntity, &group );
```

3.4 Important Device Addresses

An application may want to know the address of its device and that of its parent. Use `Zstackapi_sysNwkInfoReadReq()` to get this device’s important addresses, defined in Z-Stack TI-RTOS API [2] document, the structure return is:

```
/**
 * Structure to return the system network information read response.
 */
typedef struct _zstack_sysnwkinforeadrsp_t
{
    //! Currently assigned short address
    uint16_t nwkAddr;
    //! 64 bit IEEE Address
    zstack_LongAddr_t ieeeAddr;
    //! Current device state
    zstack_DevState devState;
    //! PAN ID
    uint16_t panId;
    //! 64 bit extended PAN ID
    zstack_LongAddr_t extendedPanId;
    //! Parent's short address
    uint16_t parentNwkAddr;
    //! 64 bit parent's extended address
    zstack_LongAddr_t parentExtAddr;
    //! possible device types
    zstack_deviceTypes_t devTypes;
    //! Current network logical channel
    uint8_t logicalChannel;
} zstack_sysNwkInfoReadRsp_t;
```

4. Binding

Binding is a mechanism to control the flow of messages from one application to another application (or multiple applications). The binding mechanism is implemented in all devices and is called source binding.

Binding allows an application to send a packet without knowing the destination address, the APS layer determines the destination address from its binding table, and then forwards the message on to the destination application (or multiple applications) or group.

4.1 Building a Binding Table

There are 3 ways to build a binding table:

- ZigBee Device Object Bind Request – a commissioning tool can tell the device to make a binding record.
- ZigBee Device Object End Device Bind Request – 2 devices can tell the coordinator that they would like to setup a binding table record. The coordinator will make the match up and create the binding table entries in the 2 devices.
- Device Application – An application on the device can build or manage a binding table.

4.1.1 ZigBee Device Object Bind Request

Any device or application can send a ZDO message to another device (over the air) to build a binding record for that other device in the network. This is called Assisted Binding and it will create a binding entry for the sending device.

4.1.1.1 The Commissioning Application

An application can do this by calling `Zstackapi_ZdoBindReq()` [defined in `zstackapi.h`] with 2 applications (addresses and endpoints) and the cluster ID wanted in the binding record. The first parameter (target `dstAddr`) is the short address of the binding's source address (where the binding record will be stored). Calling `Zstackapi_ZdoUnbindReq()` can be used, with the same parameters, to remove the binding record.

To receive the ZDO Bind or Unbind response message in your application, you will need to subscribe to response message by setting the `has_bindRsp` and `bindRsp` fields or `has_unbindRsp` and `unbindRsp` of `zstack_devZDOCBReq_t` to true and call `Zstackapi_DevZDOCBReq()`.

The target device will send back a ZigBee Device Object Bind or Unbind Response message with the status of the action.

For the Bind Response, the status returned from the coordinator will be `ZDP_SUCCESS`, `ZDP_TABLE_FULL`, `ZDP_INVALID_EP`, or `ZDP_NOT_SUPPORTED`.

For the Unbind Response, the status returned from the coordinator will be `ZDP_SUCCESS`, `ZDP_NO_ENTRY`, `ZDP_INVALID_EP`, or `ZDP_NOT_SUPPORTED`.

4.1.1.2 ZigBee Device Object End Device Bind Request

This mechanism uses a button press or other similar action at the selected devices to bind within a specific timeout period. The End Device Bind Request messages are collected at the coordinator within the timeout period and a resulting Binding Table entry is created based on the agreement of profile ID and cluster ID. The default end device binding timeout (`APS_DEFAULT_MAXBINDING_TIME`) is 16 seconds (defined in `nwk_globals.h`), but can be changed if added to `f8wConfig.cfg` or as a compile flag.

Coordinator end device binding is a toggle process. Meaning that the first time you go through the process, it will create a binding entry in the requesting devices. Then, when you go through the process again, it will remove the bindings in the requesting devices. That's why, in the following process, it will send an unbind, and wait to see if the unbind was successful. If the unbind was successful, the binding entry must have existed and been removed, otherwise it sends a binding request to make the entry.

When the coordinator receives 2 matching End Device Bind Requests, it will start the process of creating source binding entries in the requesting devices. The coordinator follows the following process, assuming matches were found in the ZDO End Device Bind Requests:

1. Send a ZDO Unbind Request to the first device. The End Device Bind is toggle process, so the unbind is sent first to remove an existing bind entry.
2. Wait for the ZDO Unbind Response, if the response status is `ZDP_NO_ENTRY`, send a ZDO Bind Request to make the binding entry in the source device. If the response status is `ZDP_SUCCESS`, move on to the cluster ID for the first device (the unbind removed the entry – toggle).
3. Wait for the ZDO Bind Response. When received, move on to the next cluster ID for the first device.
4. When the first device is done, do the same process with the second device.
5. When the second device is done, send the ZDO End Device Bind Response messages to both the first and second device.

4.1.2 Device Application Binding Manager

Another way to enter binding entries on the device is for the application to manage the binding table for itself. The application will enter and remove binding table entries locally by calling the same functions above, ZigBee Device Object Bind Request, and setting the targeted device (`nwkAddr`) to the local device's network address.

4.2 Configuring Source Binding

To enable source binding in your device include the `REFLECTOR` compile flag in `f8wConfig.cfg`. Also in `f8wConfig.cfg`, look at the 2 binding configuration items (`NWK_MAX_BINDING_ENTRIES` & `MAX_BINDING_CLUSTER_IDS`). `NWK_MAX_BINDING_ENTRIES` is the maximum number of entries in the binding table and `MAX_BINDING_CLUSTER_IDS` is the maximum number of cluster IDs in each binding entry.

The binding table is maintained in static RAM (not allocated), so the number of entries and the number of cluster IDs for each entry really affect the amount of RAM used. Each binding table entry is 6 bytes plus (`MAX_BINDING_CLUSTER_IDS * 2` bytes). Besides the amount of static RAM used by the binding table, the binding configuration items also affect the number of entries in the address manager.

5. Routing

5.1 Overview

A mesh network is described as a network in which the routing of messages is performed as a decentralized, cooperative process involving many peer devices routing on each others' behalf.

The routing is completely transparent to the application layer. The application simply sends data destined to any device down to the stack which is then responsible for finding a route. This way, the application is unaware of the fact that it is operating in a multi-hop network.

Routing also enables the "self healing" nature of ZigBee networks. If a particular wireless link is down, the routing functions will eventually find a new route that avoids that particular broken link. This greatly enhances the reliability of the wireless network and is one of the key features of ZigBee.

Many-to-one routing is a special routing scheme that handles the scenario where centralized traffic is involved. It is part of the ZigBee PRO feature set to help minimize traffic particularly when all the devices in the network are sending packets to a gateway or data concentrator. Many-to-one route discovery is described in details in Section 5.4.

5.2 Routing protocol

ZigBee uses a routing protocol that is based on the AODV (Ad-hoc On-demand Distance Vector) routing protocol for ad-hoc networks. Simplified for use in sensor networks, the ZigBee routing protocol facilitates an environment capable of supporting mobile nodes, link failures and packet losses.

Neighbor routers are routers that are within radio range of each other. Each router keeps track of their neighbors in a "neighbor table", and the "neighbor table" is updated when the router receives any message from a neighbor router (unicast, broadcast or beacon).

When a router receives a unicast packet, from its application or from another device, the NWK layer forwards it according to the following procedure. If the destination is one of the neighbors of the router (including its child devices) the packet will be transmitted directly to the destination device. Otherwise, the router will check its routing table for an entry corresponding to the routing destination of the packet. If there is an active routing table entry for the destination address, the packet will be relayed to the next hop address stored in the routing entry. If a single transmission attempt fails, the NWK layer will repeat the process of transmitting the packet and waiting for the acknowledgement, up to a maximum of `NWK_MAX_DATA_RETRIES` times. The maximum data retries in the NWK layer can be configured in `f8wconfig.cfg`. If an active entry cannot be found in the routing table or using an entry failed after the maximum number of retries, a route discovery is initiated and the packet is buffered until that process is completed.

ZigBee End Devices do not perform any routing functions. An end device wishing to send a packet to any device simply forwards it to its parent device which will perform the routing on its behalf. Similarly, when any device wishes to send a packet to an end device and initiate route discovery, the parent of the end device responds on its behalf.

Note that the ZigBee Tree Addressing (non-PRO) assignment scheme makes it possible to derive a route to any destination based on its address. In Z-Stack, this mechanism is used as an automatic fallback in case the regular routing procedure cannot be initiated (usually, due to lack of routing table space).

Also in Z-Stack, the routing implementation has optimized the routing table storage. In general, a routing table entry is needed for each destination device. But by combining all the entries for end devices of a particular parent with the entry for that parent device, storage is optimized without loss of any functionality.

ZigBee routers, including the coordinator, perform the following routing functions (i) route discovery and selection (ii) route maintenance (iii) route expiry.

5.2.1 Route Discovery and Selection

Route discovery is the procedure whereby network devices cooperate to find and establish routes through the network. A route discovery can be initiated by any router device and is always performed in regard to a particular destination device. The route discovery mechanism searches all possible routes between the source and destination devices and tries to select the best possible route.

Route selection is performed by choosing the route with the least possible cost. Each node constantly keeps track of "link costs" to all of its neighbors. The link cost is typically a function of the strength of the received signal. By adding up the link costs for all the links along a route, a “route cost” is derived for the whole route. The routing algorithm tries to choose the route with the least “route cost”.

Routes are discovered by using request/response packets. A source device requests a route for a destination address by broadcasting a Route Request (RREQ) packet to its neighbors. When a node receives an RREQ packet it in turn rebroadcasts the RREQ packet. But before doing that, it updates the cost field in the RREQ packet by adding the link cost for the latest link and makes an entry in its Route Discovery Table (5.3.2). This way, the RREQ packet carries the sum of the link costs along all the links that it traverses. This process repeats until the RREQ reaches the destination device. Many copies of the RREQ will reach the destination device traveling via different possible routes. Each of these RREQ packets will contain the total route cost along the route that it traveled. The destination device selects the best RREQ packet and sends back a Route Reply (RREP) back to the source.

The RREP is unicast along the reverse routes of the intermediate nodes until it reaches the original requesting node. As the RREP packet travels back to the source, the intermediate nodes update their routing tables to indicate the route to the destination. The Route Discovery Table, at each intermediate node, is used to determine the next hop of the RREP traveling back to the source of the RREQ and to make the entry in to the Routing Table.

Once a route is created, data packets can be sent. When a node loses connectivity to its next hop (it doesn’t receive a MAC ACK when sending data packets), the node invalidates its route by sending an RERR to all nodes that potentially received its RREP and marks the link as bad in its Neighbor Table. Upon receiving a RREQ, RREP or RERR, the nodes update their routing tables.

5.2.2 Route maintenance

Mesh networks provide route maintenance and self healing. Intermediate nodes keep track of transmission failures along a link. If a link (between neighbors) is determined as bad, the upstream node will initiate route repair for all routes that use that link. This is done by initiating a rediscovery of the route the next time a data packet arrives for that route. If the route rediscovery cannot be initiated, or it fails for some reason, a route error (RERR) packet is sent back to source of the data packet, which is then responsible for initiating the new route discovery. Either way the route gets re-established automatically.

5.2.3 Route expiry

The routing table maintains entries for established routes. If no data packets are sent along a route for a period of time, the route will be marked as expired. Expired routes are not deleted until space is needed. Thus routes are not deleted until it is absolutely necessary. The automatic route expiry time can be configured in `f8wconfig.cfg`. Set `ROUTE_EXPIRY_TIME` to expiry time in seconds. Set to 0 in order to turn off route expiry feature.

5.3 Table storage

The routing functions require the routers to maintain some tables.

5.3.1 Routing table

Each ZigBee router, including the ZigBee coordinator, contains a routing table in which the device stores information required to participate in the routing of packets. Each routing table entry contains the destination address, the next hop node, and the link status. All packets sent to the destination address are routed through the next hop node. Also entries in the routing table can expire in order to reclaim table space from entries that are no longer in use.

Routing table capacity indicates that a device routing table has a free routing table entry or it already has a routing table entry corresponding to the destination address. The routing table size is configured in `f8wconfig.cfg`. Set `MAX_RTG_ENTRIES` to the number of entries in the (default is 40). See the section on Route Maintenance for route expiration details.

5.3.2 Route discovery table

Router devices involved in route discovery, maintain a route discovery table. This table is used to store temporary information while a route discovery is in progress. These entries only last for the duration of the route discovery operation. Once an entry expires it can be used for another route discovery operation. Thus this value determines the maximum number of route discoveries that can be simultaneously performed in the network. This value is configured by setting the `MAX_RREQ_ENTRIES` in `f8wconfig.cfg`.

5.4 Many-to-One Routing Protocol

The following explains many-to-one and source routing procedure for users' better understanding of ZigBee routing protocol. In reality, all routings are taken care in the network layer and transparent to the application. Issuing many-to-one route discovery and route maintenance are application decisions.

5.4.1 Many-to-One Routing Overview

Many-to-one routing is adopted in ZigBee PRO to help minimize traffic particularly when centralized nodes are involved. It is common for low power wireless networks to have a device acting as a gateway or data concentrator. All nodes in the networks shall maintain at least one valid route to the central node. To achieve this, all nodes have to initiate route discovery for the concentrator, relying on the existing ZigBee AODV based routing solution. The route request broadcasts will add up and produce huge network traffic overhead. To better optimize the routing solution, many-to-one routing is adopted to allow a data concentrator to establish routes from all nodes in the network with one single route discovery and minimize the route discovery broadcast storm.

Source routing is part of the many-to-one routing that provides an efficient way for concentrator to send response or acknowledgement back to the destination. The concentrator places the complete route information from the concentrator to the destination into the data frame which needs to be transmitted. It minimizes the routing table size and route discovery traffic in the network.

5.4.2 Many-to-One Route Discovery

The following figure shows an example of the many-to-one route discovery procedure. To initiate many-to-one route discovery, the concentrator broadcast a many-to-one route request to the entire network. Upon receipt of the route request, every device adds a route table entry for the concentrator and stores the one hop neighbor that relays the request as the next hop address. No route reply will be generated.

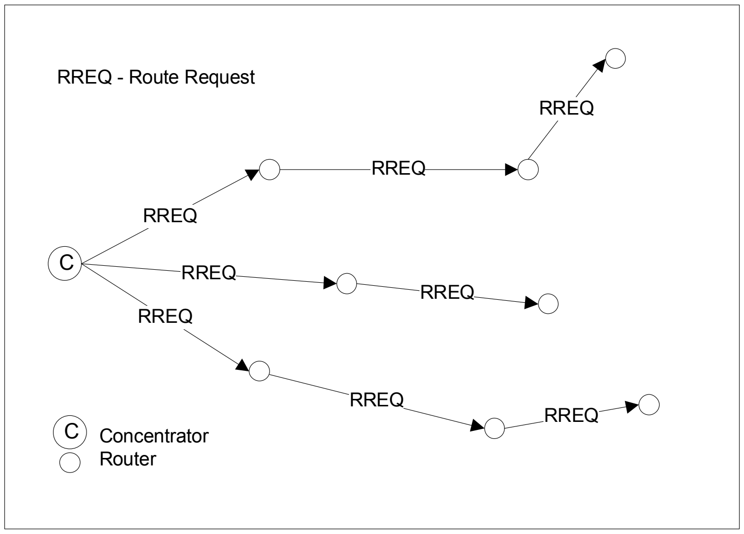


Figure 1: Many-to-one route discovery illustration

Many-to-one route request command is similar to unicast route request command with same command ID and payload frame format. The option field in route request is many-to-one and the destination address is 0xFFFC. The following Z-Stack API can be used for the concentrator to send out many-to-one route request. Please refer to the Z-Stack TI-RTOS API [2] documentation for detailed usage about this API.

```
  zstack_ZStatusValues    Zstackapi_DevNwkRouteReq(ICall_EntityID    appEntity,
                                 zstack_devNwkRouteReq_t *pReq )
```

The structure following contains the options for the route request:

```
/**
 * Structure to send a Device Network Route Request.
 */
typedef struct _zstack_devnwkroutereq_t
{
    /** Network address to discover */
    uint16_t dstAddr;
    /** True if you are announcing a concentrator */
    bool mtoRoute;
    /**
     * True if the concentrator has limited cache (only set
     * if mtoRoute is set)
     */
    bool mtoNoCache;
    /** True if the route requested is for a multicast address */
    bool multicast;
    /** Radius of the message */
    uint8_t radius;
} zstack_devNwkRouteReq_t;
```

When the `mtoRoute` and `mtoNoCache` fields are used, the `dstAddr` field will be overwritten with the many-to-one destination address 0xFFFC. Therefore, user can pass any value to `dstAddr` in the case of many-to-one route request.

5.4.3 Route Record Command

The above many-to-one route discovery procedure establishes routes from all devices to the concentrator. The reverse routing (from concentrator to other devices) is done by route record command (source routing scheme). The procedure of source routing is illustrated in Figure 2. R1 sends data packet DATA to the concentrator using the previously established many-to-one route and expects an acknowledgement back. To provide a route for the concentrator to send the ACK back, R1 sends route record command along with the data packet which records the routing path the data packet goes through and offers the concentrator a reverse path to send the ACK back.

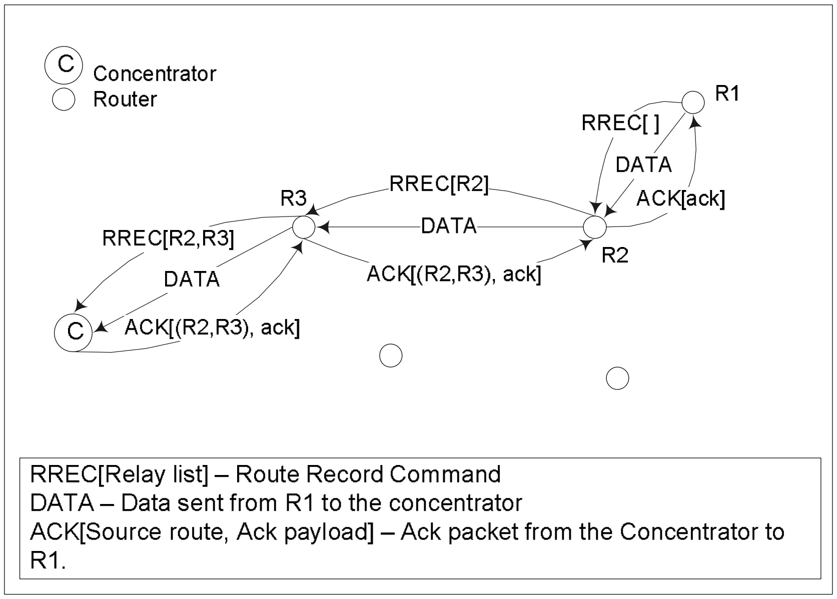

C Concentrator
Router
R1
RREC[ ]
DATA
ACK[ack]
R3
RREC[R2]
DATA
R2
RREC[R2,R3]
DATA
ACK[(R2,R3), ack]
C
ACK[(R2,R3), ack]
RREC[Relay list] – Route Record Command
DATA – Data sent from R1 to the concentrator
ACK[Source route, Ack payload] – Ack packet from the Concentrator to R1.

Figure 2: Route record command (source routing) illustration

Upon receipt of the route record command, devices on the relay path will append their own network addresses to the relay list in the route record command payload. By the time the route record command reaches the concentrator, it includes the complete routing path through which the data packet is relayed to the concentrator. When the concentrator sends ACK back to R1, it shall include the source route (relay list) in the network layer header of the packet. All devices receiving the packet shall relay the packet to the next hop device according to the source route.
For concentrator with no memory constraints, it can store all route record entries it receives and use them to send packets to the source devices in the future. Therefore, devices only need to send route record command once. However, for concentrator without source route caching capability, devices always need to send route record commands along with data packets. The concentrator will store the source route temporarily in the memory and then discard it after usage.

In brief, many-to-one routing is an efficient enhancement to the regular ZigBee unicast routing when most devices in the network are funneling traffic to a single device. As part of the many-to-one routing, source routing is only utilized under certain circumstances. First, it is used when the concentrator is responding to a request initiated by the source device. Second, the concentrator should store the source route information for all devices if it has sufficient memory. If not, whenever devices issue request to the concentrator, they should also send route record along with it.

5.4.4 Many-to-One Route Maintenance

If a link failure is encountered while a device is forwarding a many-to-one routed frame (notice that a many-to-one routed frame itself has no difference from a regular unicast data packet, however, the routing table entry has a field to specify that the destination is a concentrator), the device will generate a network status command with code "Many-to-one route failure". The network status command will be relayed to the concentrator through a random neighbor and hopefully that neighbor still has a valid route to the concentrator.

5.5 Routing Settings Quick Reference

Setting Routing Table Size	Set `MAX_RTG_ENTRIES` Note: the value must be greater than 4. (See `f8wConfig.cfg`)
Setting Route Expiry Time	Set `ROUTE_EXPIRY_TIME` to expiry time in seconds. Set to

	0 in order to turn off route expiry. (See `f8wConfig.cfg`)
Setting Route Discovery Table Size	Set `MAX_RREQ_ENTRIES` to the maximum number of simultaneous route discoveries enabled in the network. (See `f8wConfig.cfg`)
Enable Concentrator	Set `CONCENTRATOR_ENABLE` (See `ZGlobals.h`)
Setting Concentrator Property – With Route Cache	Set `CONCENTRATOR_ROUTE_CACHE` (See `ZGlobals.h`)
Setting Source Routing Table Size	Set `MAX_RTG_SRC_ENTRIES` (See `ZGlobals.h`)
Setting Default Concentrator Broadcast Radius	Set `CONCENTRATOR_RADIUS` (See `ZGlobals.h`)

6. ZDO Message Requests

The ZDO module provides functions to send ZDO service discovery request messages and receive ZDO service discovery response messages. The following flow diagram illustrates the function calls need to issue an IEEE Address Request and receive the IEEE Address Response for an application.

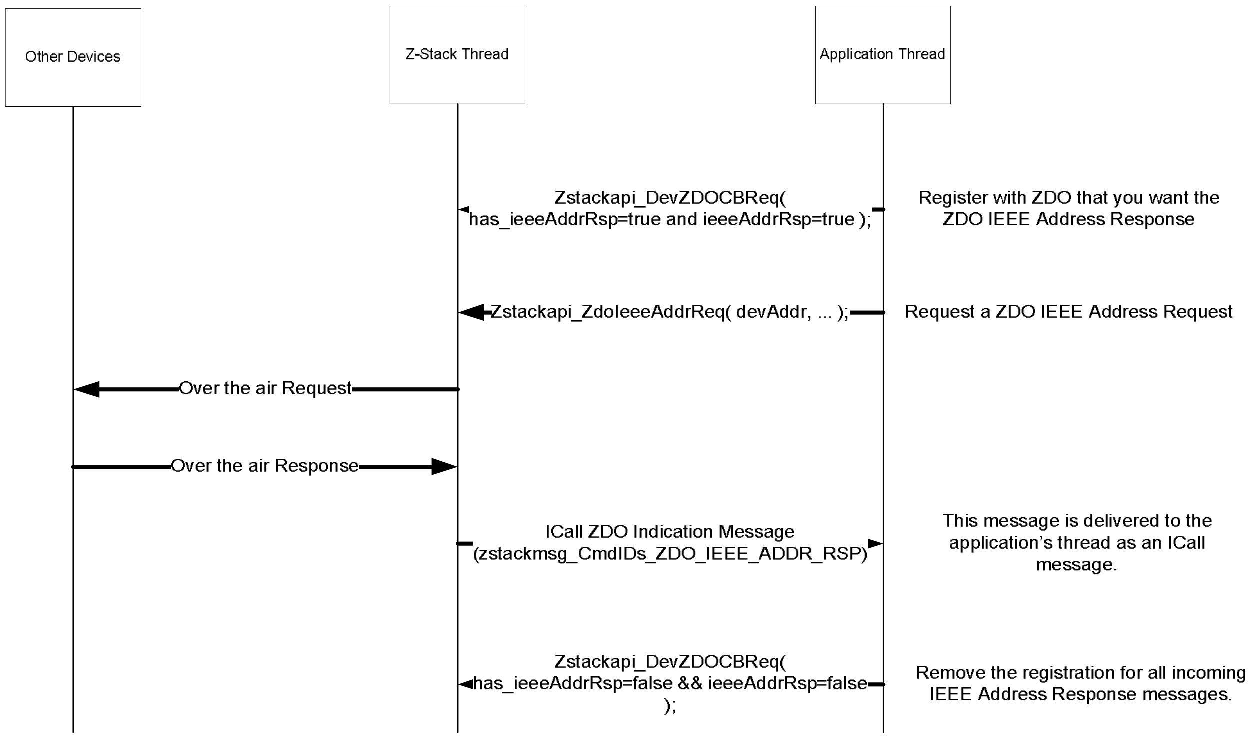

Other Devices
Z-Stack Thread
Application Thread
Zstackapi_DevZDOCBReq(has_ieeeAddrRsp=true and ieeeAddrRsp=true);
Register with ZDO that you want the ZDO IEEE Address Response
Zstackapi_ZdoIeeeAddrReq(devAddr, ...);
Request a ZDO IEEE Address Request
Over the air Request
Over the air Response
ICall ZDO Indication Message (zstackmsg_CmdIDs_ZDO_IEEE_ADDR_RSP)
This message is delivered to the application's thread as an ICall message.
Zstackapi_DevZDOCBReq(has_ieeeAddrRsp=false && ieeeAddrRsp=false);
Remove the registration for all incoming IEEE Address Response messages.

Figure 3: ZDO IEEE Address Request and Response

In the following example, an application would like to know when any new devices join the network. The application would like to receive all ZDO Device Announce (Device_annce) messages.

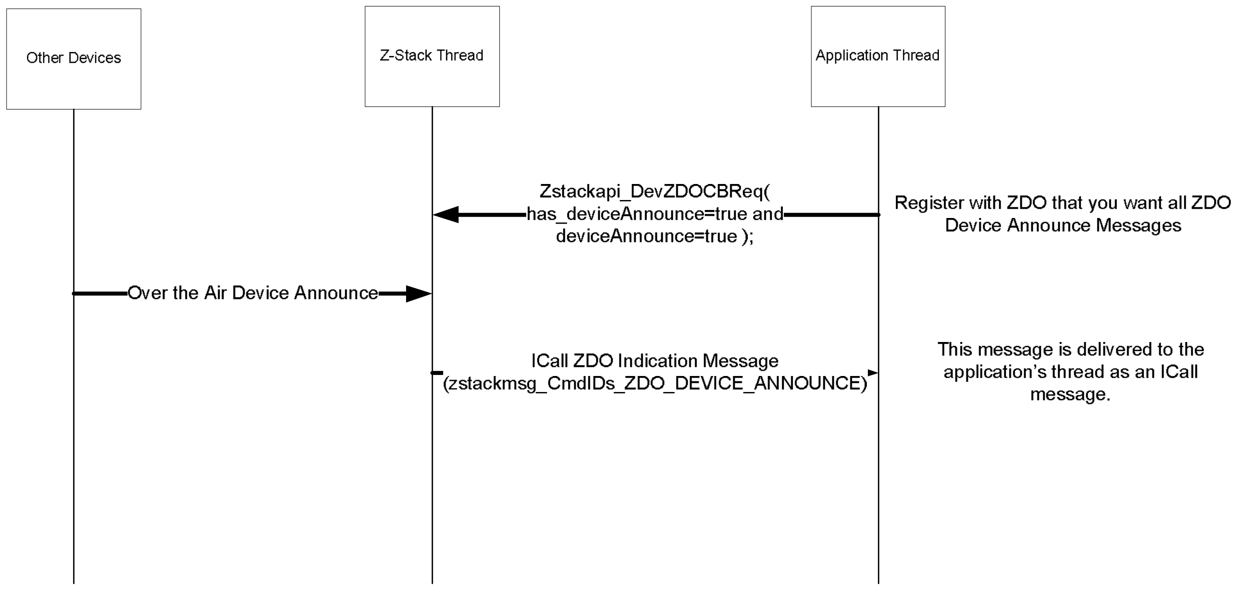

Other Devices
Z-Stack Thread
Application Thread
Zstackapi_DevZDOCBReq(has_deviceAnnounce=true and deviceAnnounce=true);
Register with ZDO that you want all ZDO Device Announce Messages
Over the Air Device Announce
ICall ZDO Indication Message (zstackmsg_CmdIDs_ZDO_DEVICE_ANNOUNCE)
This message is delivered to the application's thread as an ICall message.

Figure 4: ZDO Device Announce delivered to an application

7. Portable Devices

End devices are automatically portable. Meaning that when an end device detects that its parent isn't responding (out of range or incapacitated) it will try to rejoin the network (joining a new parent). There are no setup or compile flags to setup this option.

The end device detects that a parent isn't responding either through polling (MAC data requests) failures and/or through data message failures. The sensitivity to the failures (amount of consecutive errors) is controlled by by setting `has_pollFailureRetries = true` and `pollFailureRetries` to number of failures (the higher the number – the less sensitive and the longer it will take to rejoin), in `zstack_sysConfigWriteReq_t` in the call to `Zstackapi_sysConfigWriteReq()`.

When the network layer detects that its parent isn't responding, it will initiate a "rejoin". The rejoin process will first orphan-scan for an existing parent (the parent responds with a coordinator realignment if it recognizes the end device. If no parent is found during the orphan scan, the device will perform a network scan for a potential parent and rejoin (network rejoin command) the network with the potential parent. When searching for its parent (network discovery), the device will perform the scan on the last known active channel first, then switch to a scan on all channels.

In a secure network, it is assumed that the device already has a key and a new key isn't issued to the device.

The end device's short address is retained when it moves from parent to parent; routes to the moved end device have to be re-established automatically.

If the end device loses its parent while polling, it scans the currently active channel and attempts a secure rejoin, if it can't find its network it will attempt a scan of all channels. If attempts fail for a secure rejoin (current and all channels), the network key is removed and the device will attempt a trust center rejoin (unsecure).

When an end device is aged out, it will attempt to rejoin the network. Rejoin does not depend on state of Associate Permit flag. When the end device is in rejoin state (scan and rejoin attempts), it will be in the rejoin state for a defined period of time (15 minutes default) and then goes into a back off period(silent state) (15 minutes default), then cycle back to the rejoin state. These durations can be configured by the application image with the following code:

```
zstack_sysConfigWriteReq_t writeReq = {0};

// Set the rejoin state duration - 10 minutes (600,000 mSecs)
writeReq.has_rejoinScanDuration = true; // activate the parameter
writeReq.rejoinScanDuration = 600000;

// Set the back-off rejoin state duration - 10 minutes (600,000 mSecs)
writeReq.has_rejoinBackoffDuration = true; // activate the parameter
writeReq.rejoinBackoffDuration = 600000;

// Send the System Configuration Write Request
(void)Zstackapi_sysConfigWriteReq(appEntity, &writeReq);
```

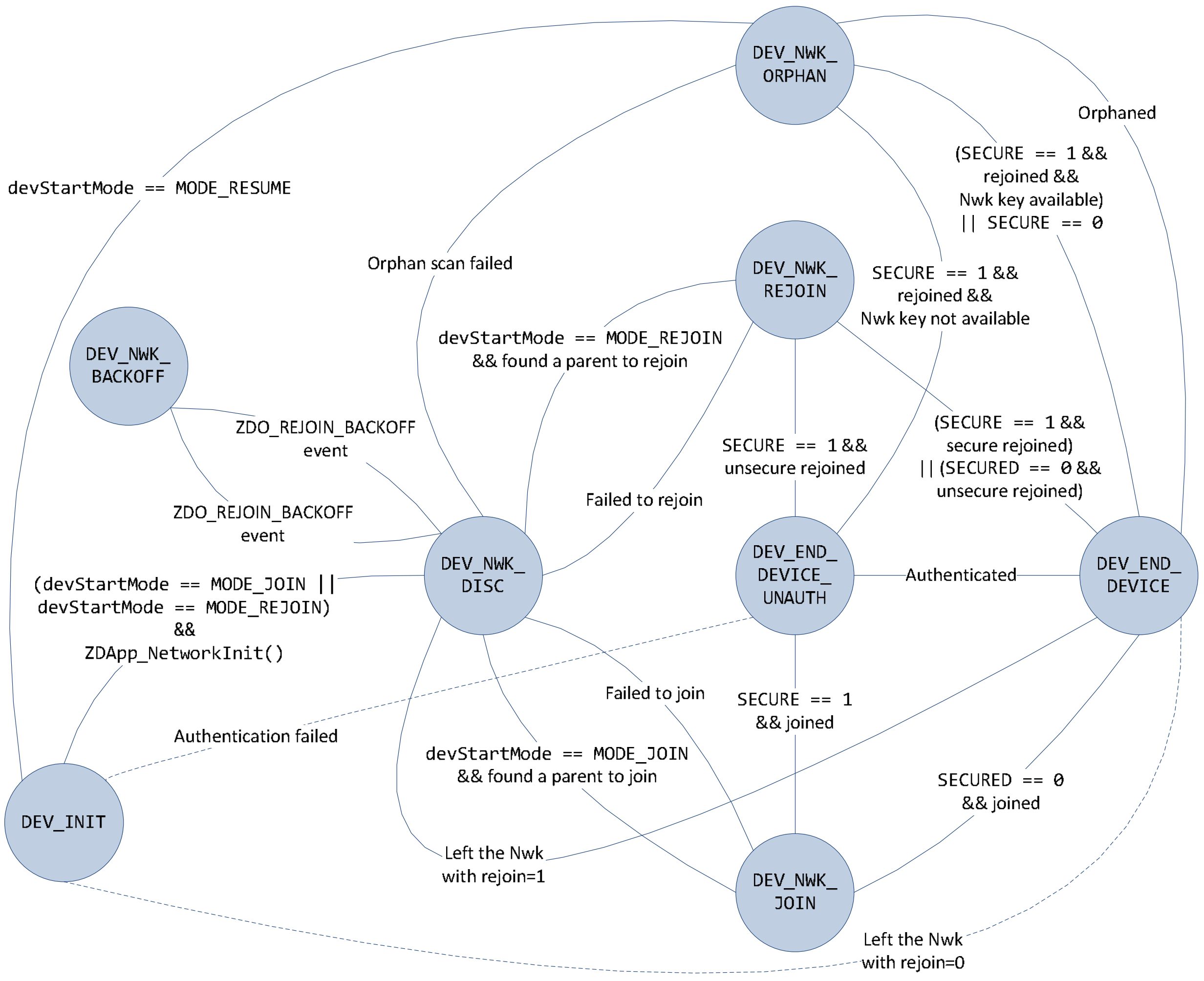

DEV_NWK_ORPHAN
DEV_NWK_REJOIN
DEV_NWK_BACKOFF
DEV_NWK_DISC
DEV_END_DEVICE_UNAUTH
DEV_END_DEVICE
DEV_INIT
DEV_NWK_JOIN
devStartMode == MODE_RESUME
Orphaned
(SECURE == 1 && rejoined && Nwk key available) || SECURE == 0
Orphan scan failed
SECURE == 1 && rejoined && Nwk key not available
devStartMode == MODE_REJOIN && found a parent to rejoin
ZDO_REJOIN_BACKOFF event
ZDO_REJOIN_BACKOFF event
SECURE == 1 && unsecure rejoined
(SECURE == 1 && secure rejoined) || (SECURED == 0 && unsecure rejoined)
Failed to rejoin
(devStartMode == MODE_JOIN || devStartMode == MODE_REJOIN) && ZDApp_NetworkInit()
Authenticated
Failed to join
SECURE == 1 && joined
Authentication failed
devStartMode == MODE_JOIN && found a parent to join
SECURED == 0 && joined
Left the Nwk with rejoin=1
Left the Nwk with rejoin=0

Figure 5. Rejoin State Diagram

8. End-to-end acknowledgements

For non-broadcast messages, there are basically 2 types of message retry: end-to-end acknowledgement (APS ACK) and single-hop acknowledgement (MAC ACK). MAC ACKs are always on by default and are usually sufficient to guarantee a high degree of reliability in the network. To provide additional reliability, as well as to enable the sending device get confirmation that a packet has been delivered to its destination, APS acknowledgements may be used.

APS acknowledgement is done at the APS layer and is an acknowledgement system from the destination device to the source device. The sending device will hold the message until the destination device sends an APS ACK message indicating that it received the message. This feature can be enabled/disabled for each message sent with the `ackRequest` field of the `options` field of the `zstack_afDataReq_t` structure when calling `Zstackapi_AfDataReq()`. The number of times that the message is retried (if APS ACK message isn't received) and the timeout between retries are configuration items in `f8wConfig.cfg`. `APSC_MAX_FRAME_RETRIES` is the number of retries the APS layer will send the message if it doesn't receive an APS ACK before giving up. `APSC_ACK_WAIT_DURATION_POLLED` is the time between retries.

9. Miscellaneous

9.1 Configuring channel

Every device must set `has_chanList=true` and `chanList`, then call `Zstackapi_sysConfigWriteReq()`, to control the channel selection. For a ZigBee Coordinator, this list will be used to scan for a channel with the least amount of noise. For ZigBee Routers and End Devices, this list will be used to scan for existing networks to join.

9.2 Configuring the PAN ID and network to join

This is an optional configuration item to control which network a ZigBee Router or End Device will join, set `has_panID=true` and `panID`, then call `Zstackapi_sysConfigWriteReq()`. A coordinator will use this value as the PAN ID of the network that it starts. A router or end device will only join a network that has a PAN ID configured in this parameter. To turn this feature off, set the parameter to a value of 0xFFFF.

9.3 Leave Network

The ZDO Management implements the function, `ZDO_ProcessMgmtLeaveReq()`[in the Z-Stack Thread], which offers access to the "NLME-LEAVE.request" primitive. The "NLME-LEAVE.request" allows a device to remove itself or remove a child device. The `ZDO_ProcessMgmtLeaveReq()` removes the device based on the provided IEEE address. If a device removes itself, it will wait for approximately 5 seconds and then reset. If a device removes a child device it will remove the device from the local "association table". The NWK address will only be reused in the case where a child device is a ZigBee End Device. In the case of a child ZigBee Router, the NWK address will not be reused.
If the parent of a child device leaves the network, the child will stay on the network.

In version R20 of the ZigBee PRO specification [1], processing of "NWK Leave Request" is configurable for Routers. The application controls this feature by setting the `zgNwkLeaveRequestAllowed` variable to `TRUE` (default value) or `FALSE`, to allow/disallow a Router to leave the network when a "NWK Leave Request" is received. `zgNwkLeaveRequestAllowed` is defined and initialized in `ZGlobals.c`, and the corresponding NV item, `ZCD_NV_NWK_LEAVE_REQ_ALLOWED`, is defined in `ZComDef.h`.

9.4 Descriptors

All devices in a ZigBee network have descriptors that describe the type of device and its applications. This information is available to be discovered by other devices in the network.

Configuration items are setup and defined in `ZDConfig.c` and `ZDConfig.h`. These 2 files also contain the Node, Power Descriptors and default User Descriptor. Make sure to change these descriptors to define your device.

9.5 Non-Volatile Memory Items

9.5.1 Global Configuration Non-Volatile Memory

Global device configuration items are stored in `ZGlobal.c`, things like PAN ID, key information, network settings. The default values for most of these items are stored in `f8wConfig.cfg`. These items are stored in RAM and accessed throughout Z-Stack. To initialize the non-volatile memory area to store these items, include the NV_INIT compile flag in your project.

9.5.2 Network Layer Non-Volatile Memory

A ZigBee device has lots of state information that needs to be stored in non-volatile memory so that it can be recovered in case of an accidental reset or power loss. Otherwise, it will not be able to rejoin the network or function effectively.

To enable this feature include the NV_RESTORE compile option. Note that this feature must usually be always enabled in a real ZigBee network and is defined by default. The ability to turn it off is only intended to be used in the development stage.
The ZDO layer is responsible for the saving and restoring of the Network Layer's vital information. This includes the Network Information Base (NIB - Attributes required to manage the network layer of the device); the list of child and parent devices, and the table containing the application bindings. Also, if security is used, some information like the frame counters will be stored.

9.6 Asynchronous Links

An asynchronous link occurs when a node can receive packets from another node but it can't send packets to that node. Whenever this happens, this link is not a good link to route packets.

In ZigBee PRO, this problem is overcome by the use of the Network Link Status message. Every router in a ZigBee PRO network sends a periodic Link Status message. This message is a one hop broadcast message that contains the sending device's neighbor list. The idea is this – if you receive your neighbor's Link Status and you are either missing from the neighbor list or your receive cost is too low (in the list), you can assume that the link between you and this neighbor is an asynchronous link and you should not use it for routing.

To change the time between Link Status messages you can change the compile flag NWK_LINK_STATUS_PERIOD, which is used to initialize _NIB.nwkLinkStatusPeriod [the _NIB is in the Z-Stack Thread]. You can also change _NIB.nwkLinkStatusPeriod directly. Remember that only PRO routers send the link status message and that every router in the network must have the same Link Status time period.

_NIB.nwkLinkStatusPeriod contains the number of seconds between Link Status messages.

Another parameter that affects the Link Status message is _NIB.nwkRouterAgeLimit (defaulted to NWK_ROUTE_AGE_LIMIT). This represents the number of Link Status periods that a router can remain in a device's neighbor list, without receiving a Link Status from that device, before it becomes aged out of the list. If we haven't received a Link Status message from a neighbor within (_NIB.nwkRouterAgeLimit * _NIB.nwkLinkStatusPeriod), we will age the neighbor out and assume that this device is missing or that it's an asynchronous link and not use it.

9.7 Multicast Messages

This feature is a ZigBee PRO only feature (must have ZIGBEEPRO as a compile flag). This feature is similar to sending to an APS Group, but at the network layer.

A multicast message is sent from a device to a group as a MAC broadcast message. The receiving device will determine if it is part of that group: if it isn't part of the group, it will decrement the non-member radius and rebroadcast; if it is part of the group it will first restore the group radius and then rebroadcast the message. If the radius is decremented to 0, the message isn't rebroadcast.

The difference between multicast and APS group messages can only be seen in very large networks where the non-member radius will limit the number of hops away from the group.

_NIB.nwkUseMultiCast [the _NIB is in the Z-Stack Thread] is used by the network layer to enable multicast (default is TRUE if ZIGBEEPRO defined) for all Group messages, and if this field is FALSE the APS Group message is sent as a normal broadcast network message.

zgApsNonMemberRadius [in the Z-Stack Thread] is the value of the group radius and the non-member radius. This variable should be controlled by the application to control the broadcast distribution. If this number is too high, the effect will be the same as an APS group message. This variable is defined in `ZGlobals.c` and `ZCD_NV_APS_NONMEMBER_RADIUS` (defined in `ZComDef.h`) is the NV item.

9.8 Fragmentation

Message Fragmentation is a process where a large message – too large to send in one APS packet – is broken down and transmitted as smaller fragments. The fragments of the larger message are then reassembled by the receiving device.

Sending a data request with a payload larger than a normal data request payload will automatically trigger fragmentation.

Fragmentation parameters can be read and set by the application by calling `Zstackapi_AfConfigGetReq()` and `Zstackapi_AfConfigSetReq()` respectively.

It is recommended that the application/profile update the `MaxInTransferSize` and `MaxOutTransferSize` of the ZDO Node Descriptor for the device, `ZDConfig_UpdateNodeDescriptor()` in `ZDConfig.c`. These fields are initialized with `MAX_TRANSFER_SIZE` (defined in `ZDConfig.h`). These values are not used in the APS layer as maximums, they are information only.

9.9 Extended PAN IDs

The application can set the extended PAN ID by calling `Zstackapi_sysConfigWriteReq()` with the `has_extendedPANID` field set to `true` and the `extendedPANID` field set to the 64-bit PAN identifier of the network to join or form (coordinator)

If the device is configured to become a ZigBee Coordinator, then it will form a network using `extendedPANID` if the value is as a non-zero value, if the value is 0x0000000000000000 (default), then the device will use its 64-bit Extended Address to form the network.

When the device is not the coordinator and `extendedPANID` has a non-zero value, then it will attempt to rejoin the network specified in `extendedPANID`. The device will join only the specified network and the procedure will fail if that network is found to be inaccessible. If `extendedPANID` is equal to 0x0000000000000000, then the device will join the best available network.

9.10 Rejoining with Pre-Commissioned Network address

During the network rejoining process a device that needs to be deployed with a predefined network address shall have configured the `zgApsUseExtendedPANID` (or as set in 9.9) and the `zgNwkCommissionedNwkAddr`, this corresponds to the `ZCD_NV_COMMISSIONED_NWK_ADDR` NV item, in the Z-Stack thread.

If configuration element `zgNwkCommissionedNwkAddr` has a valid short address value during the rejoin process, the device will put it in the `_NIB.nwkDevAddress` and use that in the Rejoin Request, otherwise it will randomly generate the short address and use it in the Rejoin Request.

10. Security

10.1 Overview

ZigBee security is built with the AES block cipher and the CCM* mode of operation as the underlying security primitive. AES/CCM* security algorithms were developed by external researchers outside of ZigBee Alliance and are also used widely in other communication protocols.

ZigBee offers the following security features:

- Infrastructure security
- Network access control
- Application data security

The Trust Center resides in the Z-Stack thread, so all of this discussion pertains to changes or settings or APIs in the Z-Stack thread project.

10.2 Configuration

In order to have a secure network, first all device images must be built with the preprocessor flag `SECURE` set equal to 1. This can be found in the `f8wConfig.cfg` file.

The default key (`defaultKey` in `nwk_globals.c`) can be preconfigured on each device in the network or it can be configured only on the coordinator and distributed to each device over-the-air as it joins the network. This is chosen via the `zgPreConfigKeys` option in `ZGlobals.c` file. If it is set to `TRUE`, then the value of default key must be preconfigured on each device (to the exact same value). If it is set to `FALSE`, then the default key parameter needs to be set only on the coordinator device. Note that in the latter case, the key will be distributed to each joining device over-air. So there is a *"moment of vulnerability"* during the joining process during which an adversary can determine the key by listening to the on-air traffic and compromise the network security.

10.3 Network access control

In a secure network, the Trust Center (coordinator) is informed when a device joins the network. The coordinator has the option of allowing that device to remain on the network or denying network access to that device.

The Trust Center may use any logic to determine if the device should be allowed into the network or not. One option is for the Trust Center to only allow devices to join during a brief time window. This may be possible, for example, if the Trust Center has a "push" button. When the button is pressed, it could allow any device to join the network for a brief time window. Otherwise all join requests would be rejected. A second possible scenario would be to configure the trust center to accept (or reject) devices based on their IEEE addresses.

This type of policy can be realized by modifying the `ZDSecMgrDeviceValidate()` function found in the `ZDSecMgr.c` module (in the Z-Stack Thread project).

10.4 Key Updates

The Trust Center can update the common Network key at its discretion. Application developers have to modify the Network key update policy. The default Trust Center implementation can be used to suit the developer's specific policy. An example policy would be to update the Network key at regular periodic intervals. Another would be to update the NWK key upon user input (like a button-press). The ZDO Security Manager `ZDSecMgr.c` API provides this functionality via `ZDSecMgrUpdateNwkKey()` and `ZDSecMgrSwitchNwkKey()`. `ZDSecMgrUpdateNwkKey()` allows the Trust Center to send a new Network key to the `dstAddr` on the network. At this point the new Network key is stored as an alternate key in the destination device or devices if `dstAddr` was a broadcast address. Once the Trust Center calls `ZDSecMgrSwitchNwkKey()`, with the `dstAddr` of the device or devices, if broadcast, all destination devices will use their alternate key.

The application thread, in a Trust Center device, can initiate key updates, and key switch by calling Zstackapi_secNwkKeyGetReq(), Zstackapi_secNwkKeySetReq(), Zstackapi_secNwkKeyUpdateReq(), and Zstackapi_secNwkKeySwitchReq(). All of these APIs can be referenced in the Z-Stack TI-RTOS API [2].

10.5 Trust Center Link Key

The ZigBee Alliance defines a default link key *ZigBeeAlliance09* in [1]. Its value is defined as `DEFAULT_TC_LINK_KEY` in `nwk_globals.h`. A different value could be used if required by the application and/or profile.

There are two types of Link Keys that can be used in a network: UNIQUE and GLOBAL. The type of Link Key used by the local device will determine how APS commands are handled as well the encryption used for those messages.

To enable all TCLK processing code (which is set by default), the `TC_LINKKEY_JOIN` compiler flag shall be defined in the Z-Stack Thread project. The application can control the type of Link Key by setting `zgApsLinkKeyType`, in `ZGlobals.h`, to value `ZG_GLOBAL_LINK_KEY` or `ZG_UNIQUE_LINK_KEY`. The corresponding NV item for `zgApsLinkKeyType is ZCD_NV_APS_LINK_KEY_TYPE`. Or, the application can call `Zstackapi_secApsLinkKeySetReq()` with the `tcLinkKey` field set to "true".

10.6 Joining a Network with TCLK

For devices that want to join a network that is using Trust Center Link Key, it is required that all devices have a pre-configured Trust Center Link Key (TCLK) and that the network key is delivered to joining devices secured with that link key. There are basically 2 joining scenarios for a joining device:

10.6.1 Multi-hop

When a device joins the network, but its parent isn't the Trust Center, the transport key command is tunneled from the Trust Center, through the parent of the joining device, to the joining device. The joining procedure is illustrated in the following figures. Notice that the APS Update Device command sent from the parent to the trust center will be encrypted according to the `zgApsLinkKeyType` configuration and using the highest APS security level. The APS Tunnel Command with APS Transport Key command as the payload is network layer encrypted but the payload is APS layer encrypted with the trust center link key between the trust center and the joining device. Finally, The APS Transport Key command forwarded from the parent to the joining device is APS encrypted with the trust center link key between the trust center and the joining device.

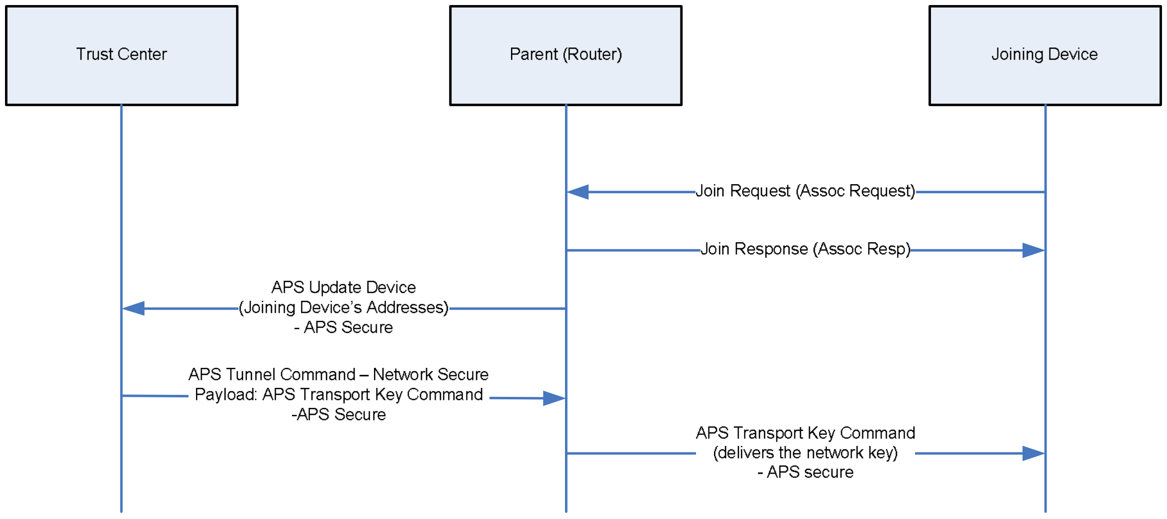

Trust Center
Parent (Router)
Joining Device
Join Request (Assoc Request)
Join Response (Assoc Resp)
APS Update Device (Joining Device's Addresses) - APS Secure
APS Tunnel Command – Network Secure Payload: APS Transport Key Command -APS Secure
APS Transport Key Command (delivers the network key) - APS secure

Figure 6: Unique Link Key Type – Joining when parent is not the Trust Center

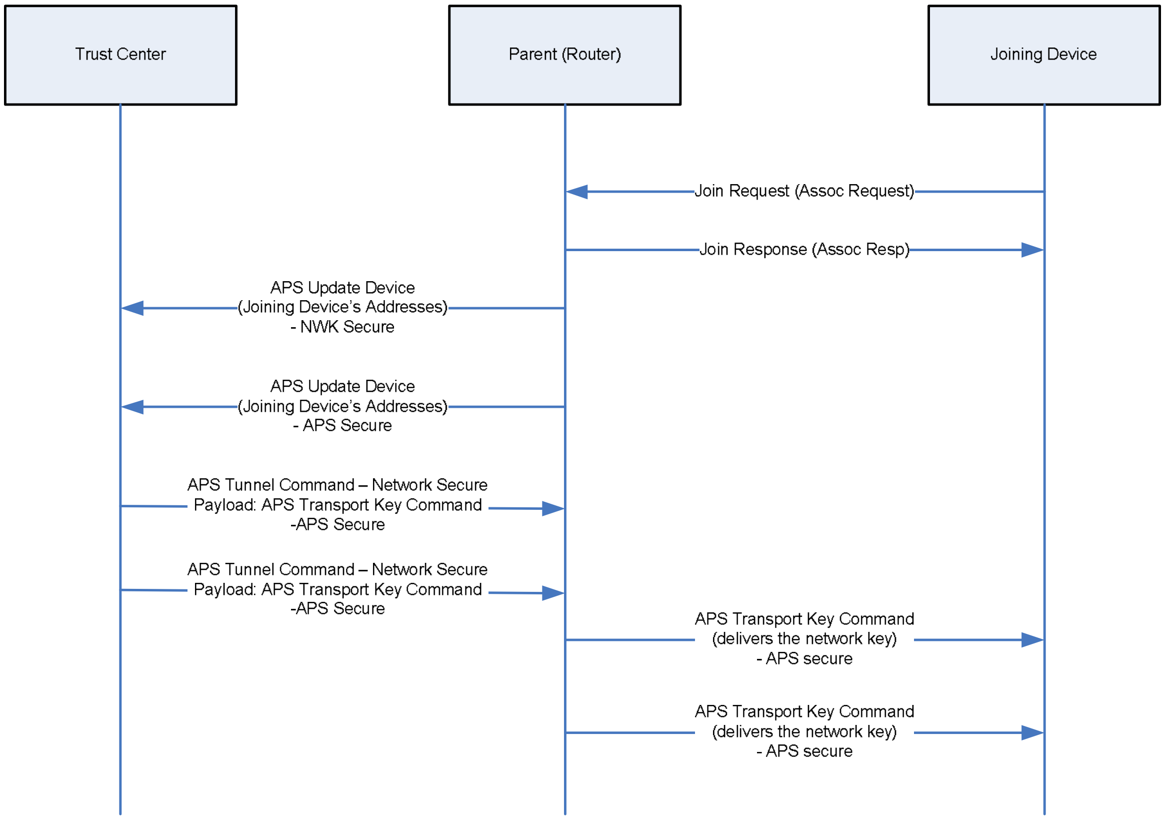

Trust Center
Parent (Router)
Joining Device
Join Request (Assoc Request)
Join Response (Assoc Resp)
APS Update Device (Joining Device's Addresses) - NWK Secure
APS Update Device (Joining Device's Addresses) - APS Secure
APS Tunnel Command – Network Secure Payload: APS Transport Key Command -APS Secure
APS Tunnel Command – Network Secure Payload: APS Transport Key Command -APS Secure
APS Transport Key Command (delivers the network key) - APS secure
APS Transport Key Command (delivers the network key) - APS secure

Figure 7: Global Link Key Type – Joining when parent is not the Trust Center

10.6.2 Single-hop

When a device joins the network, and its parent is the Trust Center, the transport key command is encrypted with the pre-configured Trust Center Link key.

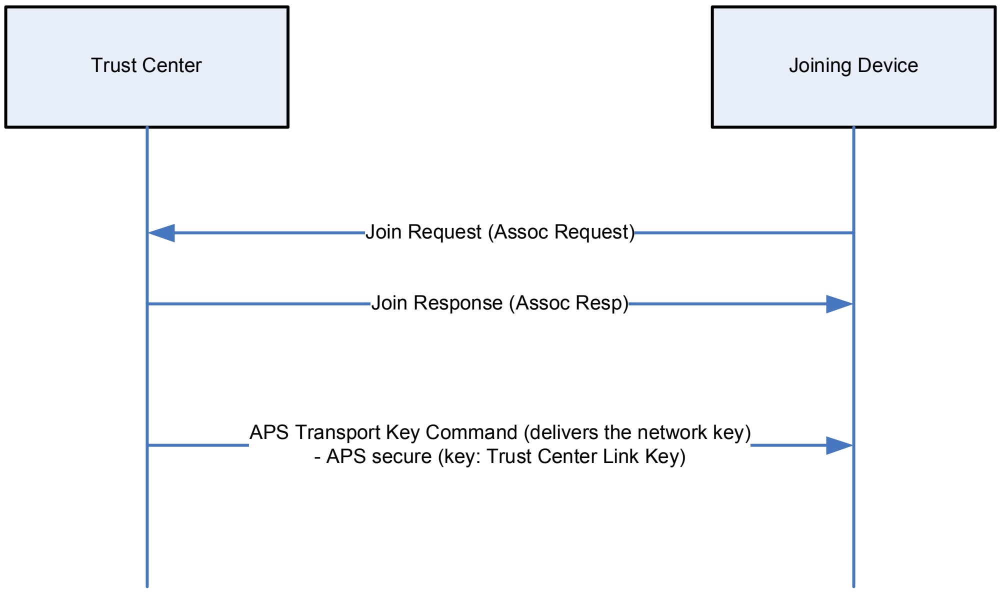

Trust Center
Joining Device
Join Request (Assoc Request)
Join Response (Assoc Resp)
APS Transport Key Command (delivers the network key)
- APS secure (key: Trust Center Link Key)

Figure 8: Global/Unique Link Key Type – Joining when parent is the Trust Center

To enable the TCLK Joining feature, set `SECURE=1` in `f8wConfig.cfg` and include the `TC_LINKKEY_JOIN` compile flag (both are set by default). There are other associated compiler flags, global variables (ZGlobals) and NV Items.

If `zgApsLinkKeyType` is set to `ZG_UNIQUE_LINK_KEY`, unique pre-configured trust center link keys are used between the Trust Center and each individual device joining the network. If `zgApsLinkKeyType` is set to `ZG_GLOBAL_LINK_KEY`, all devices are using the same pre-configured trust center link key to join the network. The Global Link Key Type provides a simplified alternative procedure to set up the network.

To start the network using Unique Link Key Type:

- Set `zgApsLinkKeyType = ZG_UNIQUE_LINK_KEY` (defined in `ZGlobals.c`), variable `zgUseDefaultTCLK` is set internally depending of this value. The NV item for these global variables are `ZCD_NV_APS_LINK_KEY_TYPE` and `ZCD_NV_USE_DEFAULT_TCLK` (defined in `ZComDef.h`).
- Set compile time option ZDSECMGR_TC_DEVICE_MAX to the maximum number of devices joining the network. Notice that it has to be no more than 255, as only 255 continuous NV ID space is reserved for preconfigured trust center link keys.
- All preconfigured trust center links keys are stored as separate NV items. The NV item ids range from `ZCD_NV_TCLK_TABLE_START` to `ZCD_NV_TCLK_TABLE_START+ ZDSECMGR_TC_DEVICE_MAX-1`. Preconfigured trust center link keys are set by configuring the NV items using SYS_OSAL_NV_WRITE for the attributes listed below:

Attribute	Description	Value
Id	NV ID for the trust center link key.	`ZCD_NV_TCLK_TABLE_START` plus an offset.
Len	Length in bytes of the item.	0x20
Offset	The memory offset into the NV item.	0x0
Value	The data array to be written to the NV item.	Its byte format is listed in the following table. All fields follow little endian first.

Table for byte format of NV item value:

Length	8 Octets	16 Octets	4 Octets	4 Octets

Attribute Field	Extended Address	Key Data	Tx Frame Counter	Rx Frame Counter
Description	Extended Address of the peer devices which shares the preconfigured tclk	The preconfigured trust center link key data	The Tx frame counter of the trust center link key	The Rx frame counter of the trust center link key

- To remove a preconfigured trust center link key, simply write all zeros to the NV item.
- It is highly recommended to erase the entire flash before using Unique Link Keys to make sure there is no existing NV item for the preconfigured trust center link keys.

To start the network using Global Link Key Type:

- Set `zgApsLinkKeyType = ZG_GLOBAL_LINK_KEY` (defined in `ZGlobals.c`), variable `zgUseDefaultTCLK` is set internally depending of this value. The NV item for these global variables are `ZCD_NV_APS_LINK_KEY_TYPE` and `ZCD_NV_USE_DEFAULT_TCLK` (defined in `ZComDef.h`).
- The default preconfigured trust center link key is written to NV item `ZCD_NV_TCLK_TABLE_START` if it has not been initialized yet. To differentiate the default preconfigured trust center link key, the extended address for default preconfigured trust center link key is all 0xFFs. The key data is initialized with `defaultTCLinkKey` (defined in `nwk_globals.c`). The Rx and Tx frame counters are initialized to all zeros.
- The default preconfigured TCLK can be changed by changing the key data, Rx and Tx frame counter fields in the NV item directly.
- It is highly recommended to erase the entire flash before using the Global Link Key Type to make sure there is no existing NV item for the default preconfigured trust center link key.
- To remove the default preconfigured trust center link key, simply write all zeros to that NV item.

Please note that the Unique Link Key and Global Link Key shall be used exclusively.

10.7 Security key data management

NV IDs for security keys are defined in `ZComDef.h` and summarized in the table below. Active and Alternate Network keys are defined as individual items, while Trust Center, Application and Master keys each reserve a range of NV IDs, allowing up to 255 keys of each type.

Value	**NV ID**	**Description**
`ZCD_NV_NWK_ACTIVE_KEY_INFO`	0x003A	Active Network key
`ZCD_NV_NWK_ALTERN_KEY_INFO`	0x003B	Alternate Network Key
`ZCD_NV_TCLK_TABLE_START`	0x0101	First element of TCLK table
`ZCD_NV_TCLK_TABLE_END`	0x01FF	Last element of TCLK table
`ZCD_NV_APS_LINK_KEY_DATA_START`	0x0201	First element of APS Link Key table
`ZCD_NV_APS_LINK_KEY_DATA_END`	0x02FF	Last element of APS Link Key table
`ZCD_NV_MASTER_KEY_DATA_START`	0x0301	First element of Master Key table
`ZCD_NV_MASTER_KEY_DATA_END`	0x03FF	Last element of Master Key table

Applications, in the application thread, can get access to link keys by calling `Zstackapi_secApsLinkKeyGetReq()` and `Zstackapi_secApsLinkKeySetReq()` in the ZStack TI-RTOS API [2].

10.8 Backwards Interoperability

There is a known interoperability issue when Unique Link Key Type is used and the Trust Center, running R20 Z-Stack, is in a network with older devices (R19). In version 20 of the ZigBee Specification [1] it is required that the Trust Center only allow APS command messages APS encrypted, but ZigBee Routers running older versions of Z-Stack send APS command messages (like Update Device) NWK encrypted only. To overcome that issue, there is a configuration control item. `zgApsAllowR19Sec` defined in `ZGlobals.c`, that the application can set to allow R19 devices to join the network. The corresponding NV item is `ZCD_NV_APS_ALLOW_R19_SECURITY` defined in `ZComDef.h`.

10.9 Quick Reference

Enabling security	Set `SECURE` = 1 (in `f8wConfig.cfg`)
Enabling preconfigured Network key	Set `zgPreConfigKeys = TRUE` (in `ZGlobals.c`)
Setting preconfigured Network key	Set `defaultKey` = {KEY} (in `nwk_globals.c`)
Enabling/disabling joining permissions on the Trust Center	Call `ZDSecMgrPermitJoining()` (in `ZDSecMgr.c`)
Specific device validation during joining	Modify `ZDSecMgrDeviceValidate` (in `ZDSecMgr.c`)
Network key updates	Call `ZDSecMgrUpdateNwkKey()` and `ZDSecMgrSwitchNwkKey()` (in `ZDSecMgr.c`)
Enabling Pre-Configured Trust Center Link Keys	Set `SECURE = 1` (in `f8wConfig.cfg`) and include `TC_LINKKEY_JOIN` or `SE_PROFILE` as a compile flag.
Use Global Trust Center Link Key	Set `zgApsLinkKeyType` = `ZG_GLOBAL_LINK_KEY` (in `ZGlobals.c`). The NV item for this global is `ZCD_NV_APS_LINK_KEY_TYPE` (defined in `ZComDef.h`).
Use Unique Trust Center Link Keys	Set `zgApsLinkKeyType` = `ZG_UNIQUE_LINK_KEY` (in `ZGlobals.c`). The NV item for this global is `ZCD_NV_APS_LINK_KEY_TYPE` (in `ZComDef.h`). Configure a preconfigured trust center link key for each device joining the network via SYS_OSAL_NV_WRITE.

11. ZMAC LQI Adjustment

11.1 Overview

The IEEE 802.15.4 specification provides some general statements on the subject of LQI. From section 6.7.8: "The minimum and maximum LQI values (0x00 and 0xFF) should be associated with the lowest and highest IEEE 802.15.4 signals detectable by the receiver, and LQI values should be uniformly distributed between these two limits." From section E.2.3: "The LQI (see 6.7.8) measures the received energy and/or SNR for each received packet. When energy level and SNR information are combined, they can indicate whether a corrupt packet resulted from low signal strength or from high signal strength plus interference."

The TI MAC computes an 8-bit "link quality index" (LQI) for each received packet from the 2.4 GHz radio. The LQI is computed from the raw "received signal strength index" (RSSI) by linearly scaling it between the minimum and maximum defined RF power levels for the radio. This provides an LQI value that is based entirely on the strength of the received signal. This can be misleading in the case of a narrowband interferer that is within the channel bandwidth – the RSSI may be increased even though the true link quality decreases.

The TI radios also provide a "correlation value" that is a measure of the received frame quality. Although not considered by the TI MAC in LQI calculation, the frame correlation is passed to the ZMAC layer (along with LQI and RSSI) in MCPS data confirm and data indication callbacks. The `ZMacLqiAdjust()` function in `zmac_cb.c` [Z-Stack Thread project] provides capability to adjust the default TI MAC value of LQI by taking the correlation into account.

11.2 LQI Adjustment Modes

LQI adjustment functionality for received frames processed in `zmac_cb.c` has three defined modes of operation - *OFF*, *MODE1*, and *MODE2*. To maintain compatibility with previous versions of Z-Stack which do not provide for LQI adjustment, this feature defaults to *OFF*, as defined by an initializer (`lqiAdjMode = LQI_ADJ_OFF;`) in `zmac_cb.c` – developers can select a different default state by changing this statement.

MODE1 provides a simple algorithm to use the packet correlation value (related to SNR) to scale incoming LQI value (related to signal strength) to 'de-rate' noisy packets. The incoming LQI value is linearly scaled with a "correlation percentage" that is computed from the raw correlation value between theoretical minimum/maximum values (`LQI_CORR_MIN` and `LQI_CORR_MAX` are defined in `ZMAC.h`).

MODE2 provides a "stub" for developers to implement their own proprietary algorithm. Code can be added after the "`else if ( lqiAdjMode == LQI_ADJ_MODE2 )`" statement in `ZMacLqiAdjust()`.

11.3 Using LQI Adjustment

There are two ways to enable the LQI adjustment functionality:

(1) Alter the initialization of the `lqiAdjMode` variable as described in the previous section
(2) Call the function `ZMacLqiAdjustMode()` from somewhere within the Z-Stack application, most likely from the application's task initialization function. See the Z-Stack API **Error! Reference source not found.** document on details of this function.

The `ZMacLqiAdjustMode()` function can be used to change the LQI adjustment mode as needed by the application. For example, a developer might want to evaluate device/network operation using a proprietary *MODE2* compared to the default *MODE1* or *OFF*.

Tuning of *MODE1* operation can be achieved by altering the values of LQI_CORR_MIN and/or LQI_CORR_MAX. When using IAR development tools, alternate values for these parameters can be provided as compiler directives in the IDE project file or in one of Z-Stack's .cfg files (`f8wConfig.cfg`, `f8wCoord.cfg`, etc.). Refer to the radio's data sheet for information on the normal minimum/maximum correlation values.

12. Compile Options

12.1 Overview

This section provides information and procedures for using compiler options with Texas Instruments Z-Stack™, and it's recommend that you don't change compile flags that aren't listed in this section.

12.2 Requirements

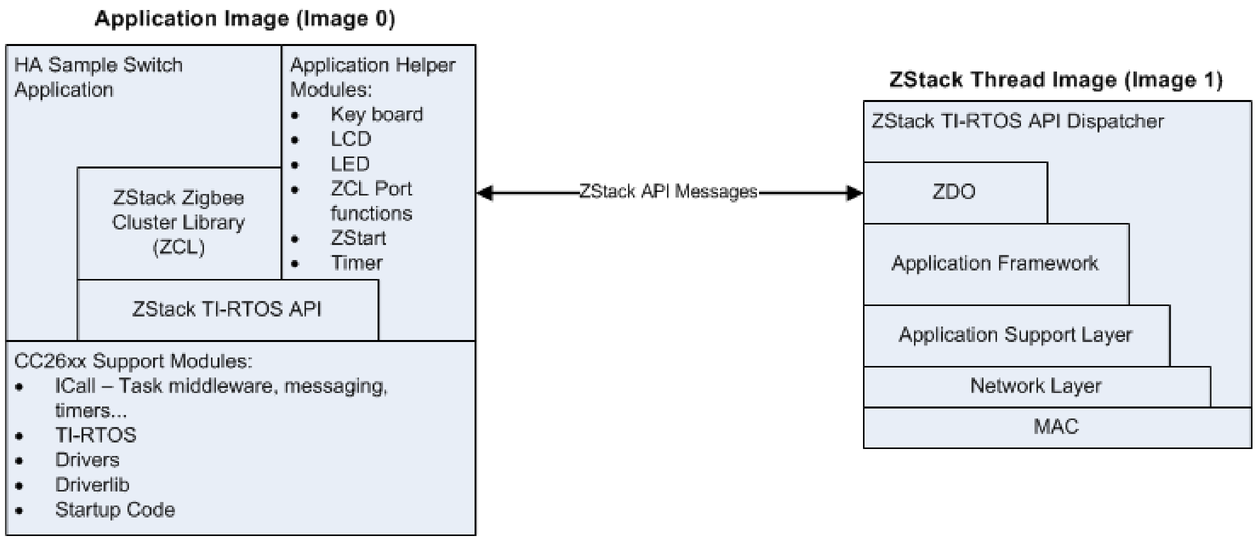
12.2.1 Target Development System Requirements

Z-Stack is built on top of the IAR Embedded Workbench (IAR EWARM) suite of software development tools (www.iar.com). These tools support project management, compiling, assembling, linking, downloading, and debugging for various development platforms.

Each of the supplied sample applications contain a IAR workspace (ex. GenericApp.eww). The workspace contains 2 projects:

- The sample application (ex. GenericApp.ewp) – contains the sample application, Zigbee cluster library (if needed), drivers, RTOS and startup code.
- The ZStack Core Thread – contains the Zigbee specific layers (ZDO, AF, APS, NWK and MAC).

The 2 projects communicate through a messaging interface called "ICall". The ZStack TI-RTOS API [2] is a set of APIs that wrap the message into functions (defined in zstackapi.h) the application can call to communicate with the Z-Stack Thread.


Application Image (Image 0)
HA Sample Switch Application
Application Helper Modules:
Key board
LCD
LED
ZCL Port functions
ZStart
Timer
ZStack Zigbee Cluster Library (ZCL)
ZStack TI-RTOS API
CC26xx Support Modules:
ICall – Task middleware, messaging, timers...
TI-RTOS
Drivers
Driverlib
Startup Code
ZStack API Messages
ZStack Thread Image (Image 1)
ZStack TI-RTOS API Dispatcher
ZDO
Application Framework
Application Support Layer
Network Layer
MAC

The images or projects are downloaded to the device, usually in the following steps:

1. Erase the flash (Project->Download->Erase Memory) to clear the entire flash space (NV included).
2. Program the Z-Stack Thread Image, example: select the ZStackCore-EndDevice project pull down the project menu, select Download, select Download active application (Project->Download->Download active application).
3. Program and debug the application image. Example, select the GenericApp-SmartRF06 project, pull down the project menu, select Download and Debug (Project->Download and Debug)

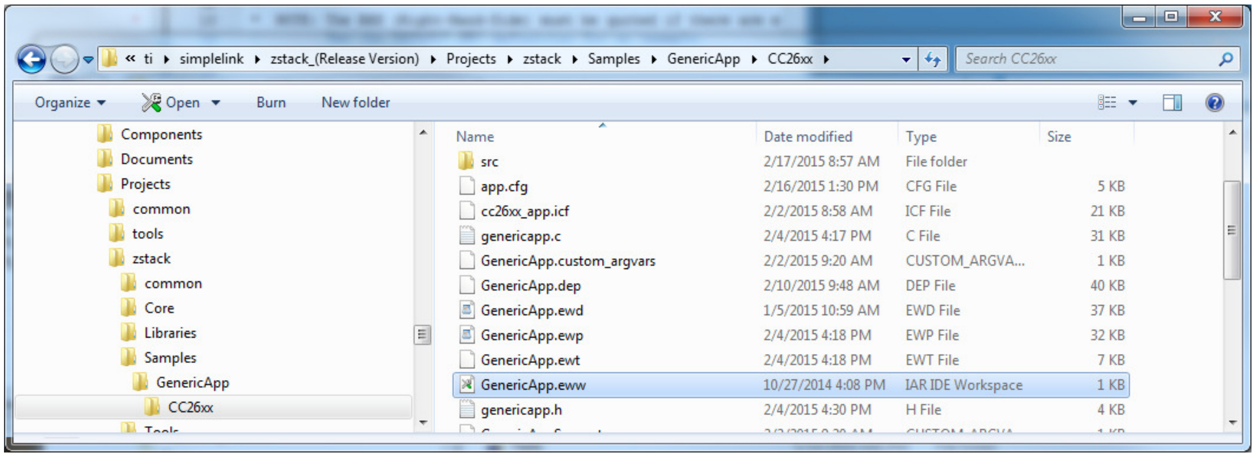
12.3 Using Compile Options

12.3.1 Locating Compile Options

Compile options for a specific project are located in two places. Options that are rarely, if ever, changed are located in linker control files, one for each logical device type discussed above. User-defined options and ones that change to enable/disable features are located in the IAR project file. For demonstration purposes, these two files for the GenericApp Coordinator project will be examined. Access to all other Z-Stack projects will be similar.

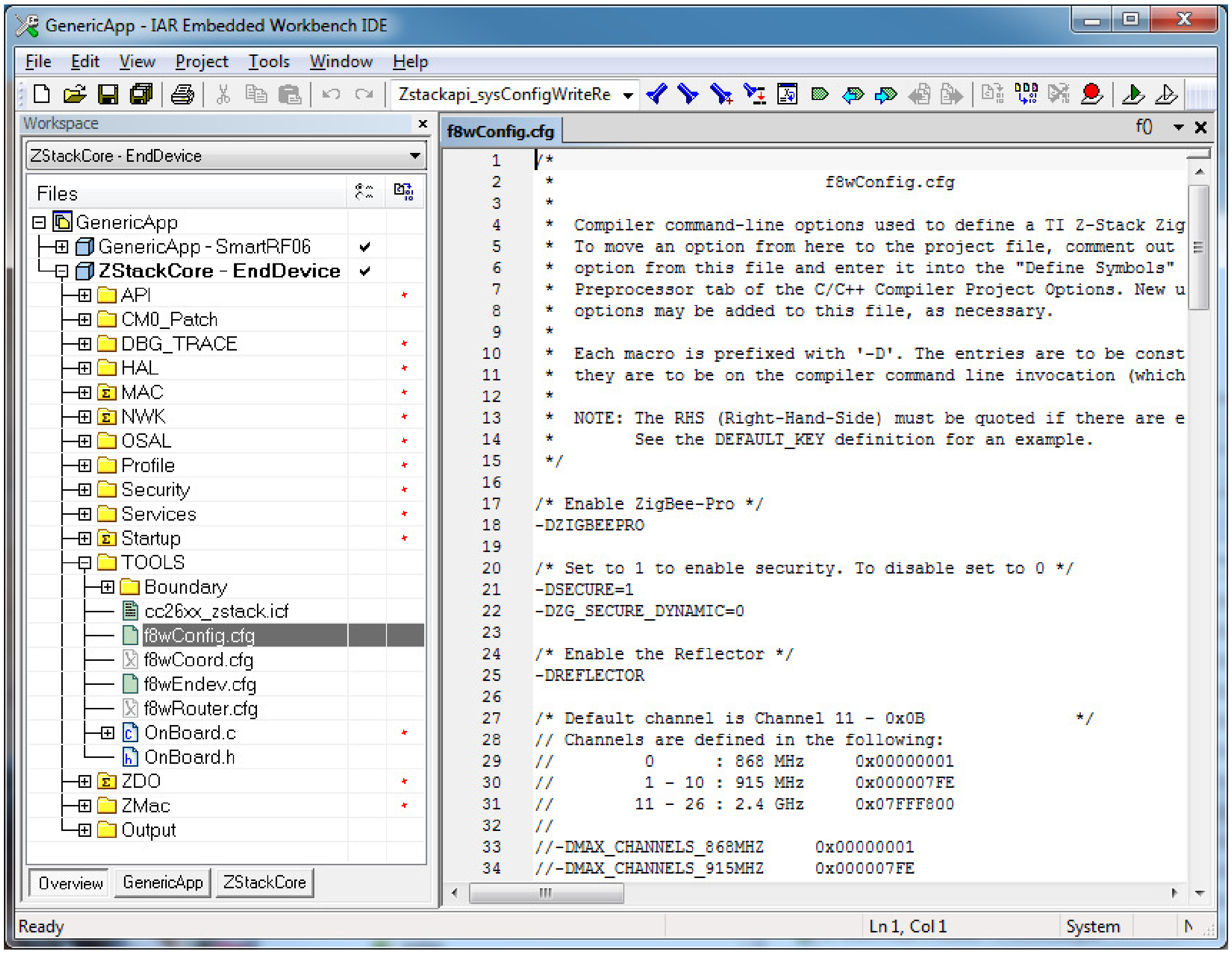
12.3.1.1 Compile Options In Linker Control Files

GenericApp project files are found in the **..\Projects\zstack\Samples\GenericApp\(Platform)** folder:



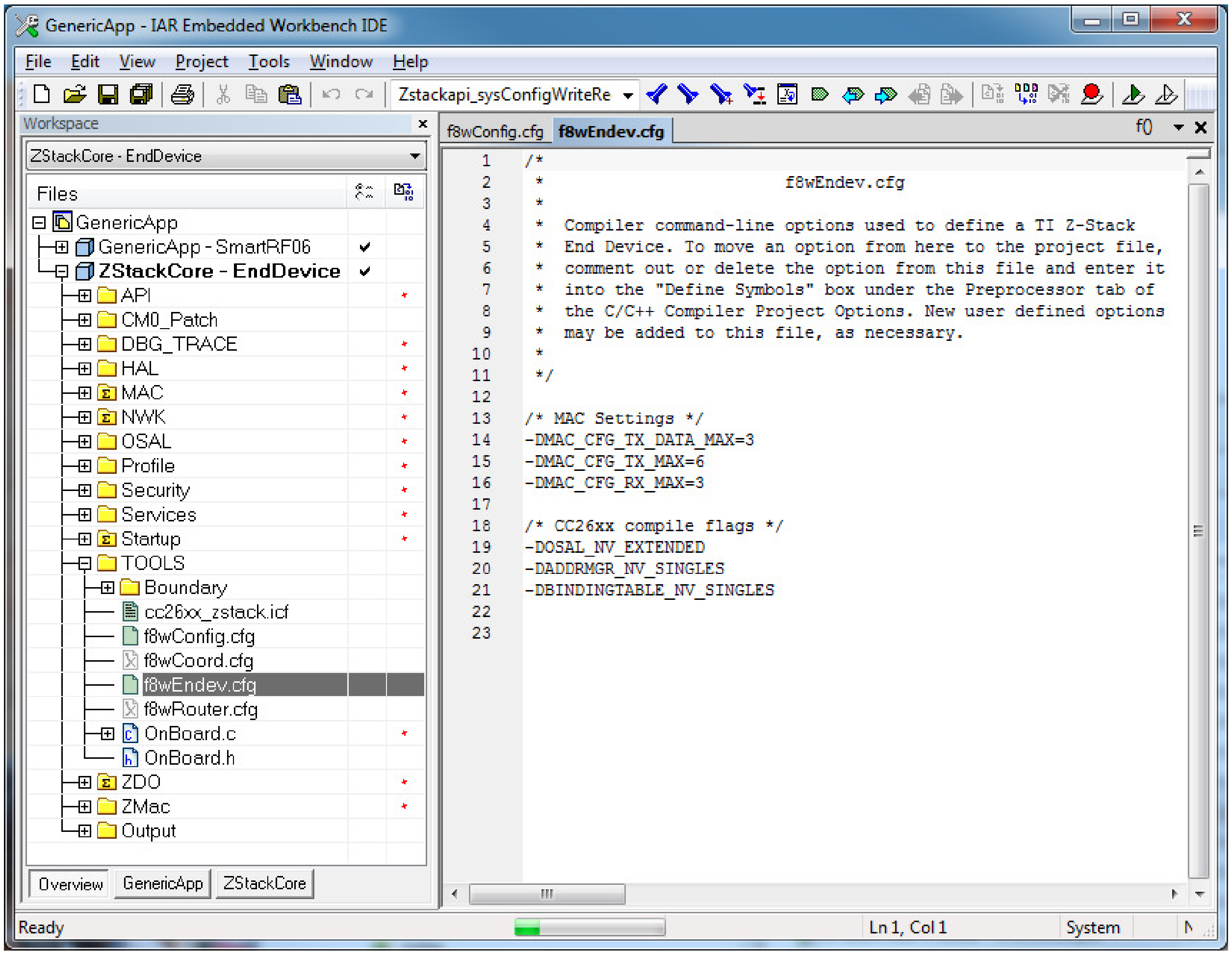
Name	Date modified	Type	Size
src	2/17/2015 8:57 AM	File folder	
app.cfg	2/16/2015 1:30 PM	CFG File	5 KB
cc26xx_app.icf	2/2/2015 8:58 AM	ICF File	21 KB
genericapp.c	2/4/2015 4:17 PM	C File	31 KB
GenericApp.custom_argvars	2/2/2015 9:20 AM	CUSTOM_ARGVA...	1 KB
GenericApp.dep	2/10/2015 9:48 AM	DEP File	40 KB
GenericApp.ewd	1/5/2015 10:59 AM	EWD File	37 KB
GenericApp.ewp	2/4/2015 4:18 PM	EWP File	32 KB
GenericApp.ewt	2/4/2015 4:18 PM	EWT File	7 KB
GenericApp.eww	10/27/2014 4:08 PM	IAR IDE Workspace	1 KB
genericapp.h	2/4/2015 4:30 PM	H File	4 KB

Open the workspace by double-clicking on the *GenericApp.eww* file, the workspace contains 2 projects, each representing separate images: the application image and the Z-Stack Core image. Select the *ZStackCore -EndDevice* project, then open the **Tools** folder. Several linker control files are located in the **Tools** folder. This folder contains various configuration files and executable tools used in Z-Stack projects. Generic compile options are defined in the `f8wConfig.cfg` file. This file, for example, specifies the channel(s) and the PAN ID that will be used when a device starts up. This is the recommended location for a user to establish specific channel settings for their projects. This allows developers set up "personal" channels to avoid conflict with others. Device specific compile options are located in the `f8wCoord.cfg`, `f8wEndev.cfg`, and `f8wRouter.cfg` files:



```
/*
 *                              f8wConfig.cfg
 *
 *  Compiler command-line options used to define a TI Z-Stack Zig
 *  To move an option from here to the project file, comment out
 *  option from this file and enter it into the "Define Symbols"
 *  Preprocessor tab of the C/C++ Compiler Project Options. New u
 *  options may be added to this file, as necessary.
 *
 *  Each macro is prefixed with '-D'. The entries are to be const
 *  they are to be on the compiler command line invocation (which
 *
 *  NOTE: The RHS (Right-Hand-Side) must be quoted if there are e
 *        See the DEFAULT_KEY definition for an example.
 */

/* Enable ZigBee-Pro */
-DZIGBEEPRO

/* Set to 1 to enable security. To disable set to 0 */
-DSECURE=1
-DZG_SECURE_DYNAMIC=0

/* Enable the Reflector */
-DREFLECTOR

/* Default channel is Channel 11 - 0x0B             */
// Channels are defined in the following:
//         0       : 868 MHz     0x00000001
//         1 - 10  : 915 MHz     0x000007FE
//        11 - 26  : 2.4 GHz     0x07FFF800
//
//-DMAX_CHANNELS_868MHZ     0x00000001
//-DMAX_CHANNELS_915MHZ     0x000007FE
```

The GenericApp EndDevice project uses the `f8wEndev.cfg` file. As shown below, compile options that are specific to end devices and options that provide "generic" Z-Stack functions are included in this file:



```
/*
 *                          f8wEndev.cfg
 *
 *  Compiler command-line options used to define a TI Z-Stack
 *  End Device. To move an option from here to the project file,
 *  comment out or delete the option from this file and enter it
 *  into the "Define Symbols" box under the Preprocessor tab of
 *  the C/C++ Compiler Project Options. New user defined options
 *  may be added to this file, as necessary.
 *
 */

/* MAC Settings */
-DMAC_CFG_TX_DATA_MAX=3
-DMAC_CFG_TX_MAX=6
-DMAC_CFG_RX_MAX=3

/* CC26xx compile flags */
-DOSAL_NV_EXTENDED
-DADDRMGR_NV_SINGLES
-DBINDINGTABLE_NV_SINGLES
```

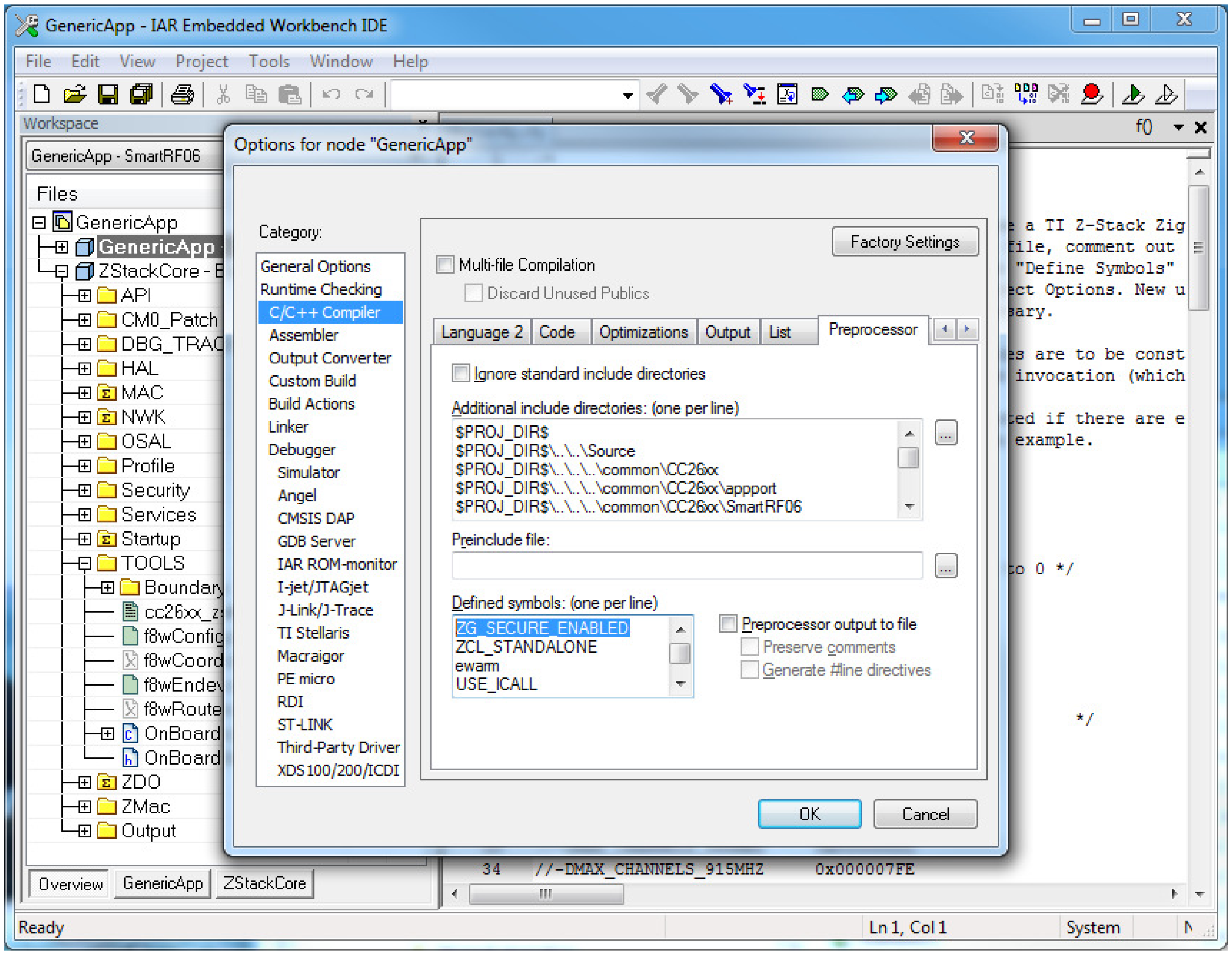
The `f8wCoord.cfg` file is used by all projects that build Coordinator devices. Therefore, any change made to this file will affect all Coordinators. In a similar manner, the `f8wRouter.cfg` and `f8wEnd.cfg` files affect all Router and End-Device projects, respectively.

To add a compile option to all projects of a certain device type, simply add a new line to the appropriate linker control file. To disable a compile option, comment that option out by placing // at the left edge of the line. You could also delete the line but this is not recommended since the option might need to be re-enabled at a later time.

12.3.1.2 Compile Options in IAR Project Files

The compile options for each of the supported configurations are stored in the *GenericApp.ewp and ZStackCore.ewp* files. To modify these compile options, first select **project (GenericApp-SmartRF06 or ZStackCore-EndDevice)**. Then select the **Options…** item from the **Project** pull-down menu.

Select the **C/C++ Compiler** item and click on the **Preprocessor** tab. The compile options for this configuration are located in the box labeled *Defined symbols: (one per line)*:

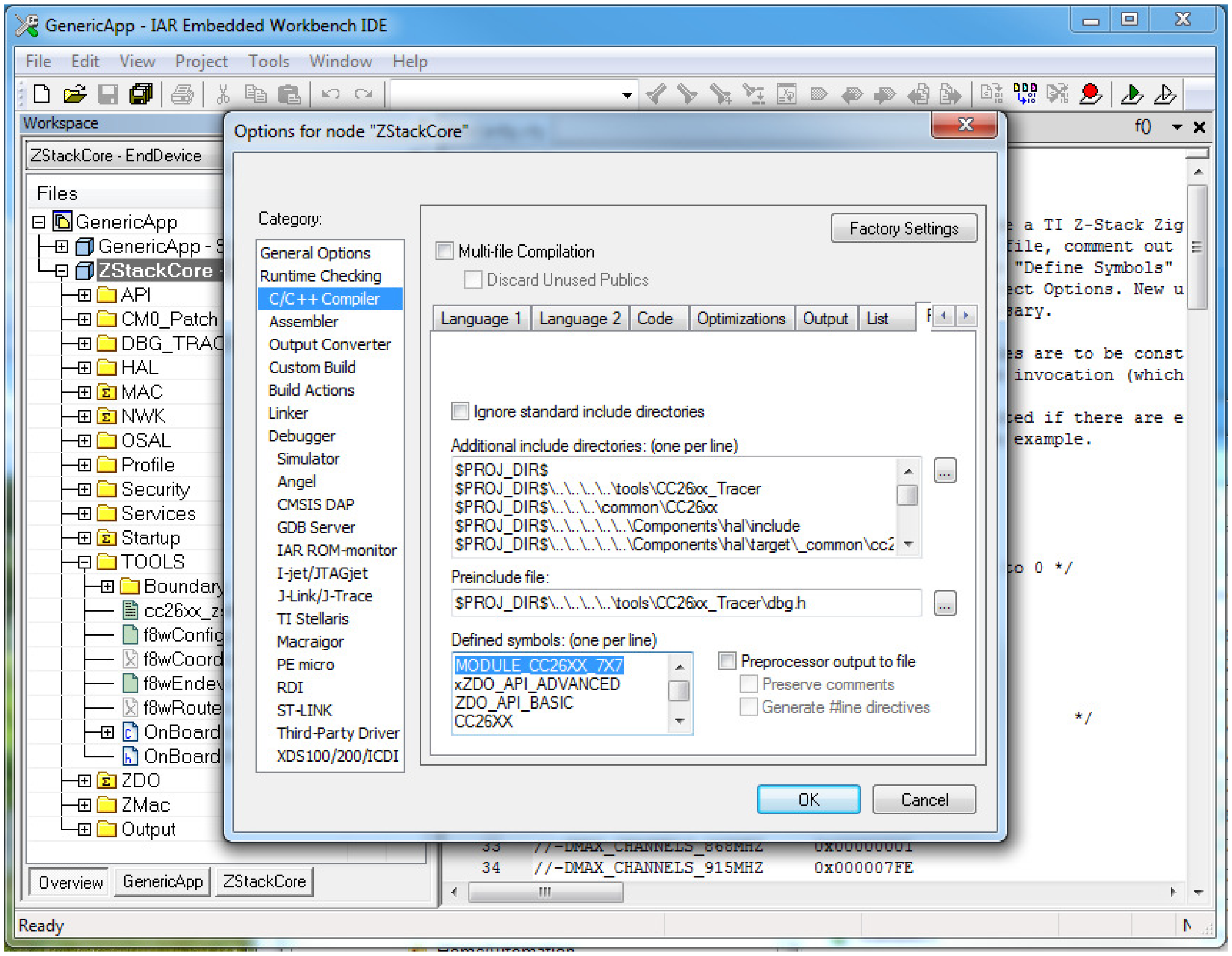


Additional include directories: (one per line)

```
$PROJ_DIR$
$PROJ_DIR$\..\..\Source
$PROJ_DIR$\..\..\..\common\CC26xx
$PROJ_DIR$\..\..\..\common\CC26xx\appport
$PROJ_DIR$\..\..\..\common\CC26xx\SmartRF06
```

Defined symbols: (one per line)

```
ZG_SECURE_ENABLED
ZCL_STANDALONE
ewarm
USE_ICALL
```



Options for node "ZStackCore"

Category:
General Options
Runtime Checking
C/C++ Compiler
Assembler
Output Converter
Custom Build
Build Actions
Linker
Debugger
Simulator
Angel
CMSIS DAP
GDB Server
IAR ROM-monitor
I-jet/JTAGjet
J-Link/J-Trace
TI Stellaris
Macraigor
PE micro
RDI
ST-LINK
Third-Party Driver
XDS100/200/ICDI

Factory Settings

Multi-file Compilation
Discard Unused Publics

Language 1 | Language 2 | Code | Optimizations | Output | List

Ignore standard include directories

Additional include directories: (one per line)

```
$PROJ_DIR$
$PROJ_DIR$\..\..\..\..\tools\CC26xx_Tracer
$PROJ_DIR$\..\..\..\common\CC26xx
$PROJ_DIR$\..\..\..\..\..\Components\hal\include
$PROJ_DIR$\..\..\..\..\..\Components\hal\target\_common\cc2
```

Preinclude file:

```
$PROJ_DIR$\..\..\..\..\tools\CC26xx_Tracer\dbg.h
```

Defined symbols: (one per line)

```
MODULE_CC26XX_7X7
xZDO_API_ADVANCED
ZDO_API_BASIC
CC26XX
```

Preprocessor output to file
Preserve comments
Generate #line directives

OK Cancel

To add a compile option to this configuration, simply add the item on a new line within this box. To disable a compile option, place an ‘**x**’ at the left edge of the line. Note that ZDO_API_ADVANCED has an ‘x’ in front of it, this option could have been deleted but this is not recommended since it might need to be re-enabled at a later time.

12.3.2 Using Compile Options

Compile options are used to select features that are provided in the source files. Most compile options act as on/off switches for specific sections within source programs. Some options are used to provide a user-defined numerical value, such as `DEFAULT_CHANLIST`, to the compiler to override default values.

Each of the Z-Stack sample applications (ex. GenericApp) provide an IAR project file which specifies the compile options to be used for that specific project. The programmer can add or remove options as needed to include or exclude portions of the available software functions.

The next sections of this document provide lists of the supported compile options with a brief description of what feature they enable or disable. Options that are listed as “do not change” are required for proper operation of the compiled programs. Options that are listed as “*do not use*” are not appropriate for use with the board.

12.4 Supported Compile Options and Definitions

12.4.1 General Compile Options for the Z-Stack Thread poject

The compile options in the following table can be changed or set to select desired features, a lot of these compile options are set and described in `f8wConfig.cfg`.

APS_DEFAULT_INTERFRAME_DELAY	Delay between Tx packets when using fragmentation
APS_DEFAULT_MAXBINDING_TIME	Maximum time in seconds that a Coordinator will wait between receiving match descriptor bind requests to perform binding
APS_DEFAULT_WINDOW_SIZE	Size of a Tx window when using fragmentation
APS_MAX_GROUPS	Maximum number of entries allowed in the groups table
APSC_ACK_WAIT_DURATION_POLLED	Number of 2 milliseconds periods a polling End Device will wait for an APS acknowledgement from the destination device
APSC_MAX_FRAME_RETRIES	Maximum number of retries allowed (at APS layer) after a transmission failure
ASSERT_RESET	Specifies that the device should reset when there's an assertion. When not defined, all LEDs will flash when an assertion occurs.
BEACON_REQUEST_DELAY	Minimum number of milliseconds to delay between each beacon request in a joining cycle
DEFAULT_CHANLIST	Change this list in f8wConfig.cfg
EXTENDED_JOINING_RANDOM_MASK	Mask for the random joining delay
HOLD_AUTO_START	Disable automatic start-up of ZDApp event processing loop
LCD_SUPPORTED	Enable LCD emulation – text sent to ZTool serial port
MANAGED_SCAN	Enable delays between channel scans
MAX_BCAST	Maximum number of simultaneous broadcasts supported by a device at any given time
MAX_BINDING_CLUSTER_IDS	Maximum number of cluster IDs in a binding record
MAX_POLL_FAILURE_RETRIES	Number of times retry to poll parent before indicating loss of synchronization with parent. Note that larger value will cause longer delay for the child to rejoin the network
MAX_RREQ_ENTRIES	Number of simultaneous route discoveries in network
MAX_RTG_ENTRIES	Number of entries in the regular routing table plus additional entries for route repair
MAXMEMHEAP	Determines the total memory available for dynamic memory. Every request for an amount of dynamic memory requires dynamic memory space for overhead used in managing the allocated memory. So MAXMEMHEAP does not reflect the total amount of dynamic memory that the user can expect to be usable. As a rule of thumb, each memory allocation requires at least 2+N bytes, where N represents the word-alignment block size of the target CPU (e.g., N=1 on the AVR and CC2430 but N=2 on the MSP430). MAXMEMHEAP must be defined to be less that 32768
NV_INIT	Enable loading of "basic" NV items at device reset
NV_RESTORE	Enables device to save/restore network state information to/from NV
NWK_AUTO_POLL	Enable End Device to poll from the parents automatically
NWK_INDIRECT_MSG_TIMEOUT	Number of milliseconds the parent of a polling End Device will hold a message
NWK_MAX_BINDING_ENTRIES	Maximum number of entries in the binding table
NWK_MAX_DATA_RETRIES	The maximum number of times retry looking for the next hop address of a message
NWK_MAX_DEVICE_LIST	Maximum number of devices in the Association/Device list
NWK_MAX_DEVICES	Maximum number of devices in the network
NWK_START_DELAY	Minimum number of milliseconds to hold off the start of the device in the

	network and the minimum delay between joining cycles
OSAL_TOTAL_MEM	Track OSAL memory heap usage (display if LCD_SUPPORTED)
POLL_RATE	For end devices only: number of milliseconds to wait between data request polls to its parent. Example POLL_RATE=1000 is a data request every second. This is changed in *f8wConfig.cfg*.
POWER_SAVING	Enable power saving functions for battery-powered devices
QUEUED_POLL_RATE	This is used after receiving a data indication to poll immediately for queued messages (in milliseconds)
REFLECTOR	Enable binding
REJOIN_POLL_RATE	This is used as an alternate response poll rate only for rejoin request. This rate is determined by the response time of the parent that the device is trying to join
RESPONSE_POLL_RATE	This is used after receiving a data confirmation to poll immediately for response messages (in milliseconds)
ROUTE_EXPIRY_TIME	Number of seconds before an entry expires in the routing table; set to 0 to turn off route expiry
RTR_NWK	Enable Router networking
SECURE	Enable ZigBee security (SECURE=0 to disable, SECURE=1 to enable)
ZDAPP_CONFIG_PAN_ID	Coordinator’s PAN ID; used by Routers and End Devices to join PAN with this ID
ZDO_COORDINATOR	Enable the device as a Coordinator
ZIGBEEPRO	Enable usage of ZigBee Pro features

12.4.2 ZigBee Device Object (ZDO) Compile Options

By default, the mandatory messages (as defined by the ZigBee spec) are enabled in the ZDO. All other message processing is controlled by compile flags. You can enable/disable the options by commenting/un-commenting the compile flags in `ZDConfig.h` or include/exclude them like other compile flags. There's an easy way to enable all the ZDO Function and Management options: You can use `ZDO_API_ADVANCED` to enable all the ZDO Function options, and `ZDO_API_BASIC` to enable all the ZDO Functions used in all the sample applications. Information about the use of these messages is provided in this guide and Z-Stack API document.

ZDO_NWKADDR_REQUEST	Enable Network Address Request function and response processing
ZDO_IEEEADDR_REQUEST	Enable IEEE Address Request function and response processing
ZDO_MATCH_REQUEST	Enable Match Descriptor Request function and response processing
ZDO_NODEDESC_REQUEST	Enable Node Descriptor Request function and response processing
ZDO_POWERDESC_REQUEST	Enable Power Descriptor Request function and response processing
ZDO_SIMPLEDESC_REQUEST	Enable Simple Descriptor Request function and response processing
ZDO_ACTIVEEP_REQUEST	Enable Active Endpoint Request function and response processing
ZDO_COMPLEXDESC_REQUEST	Enable Complex Descriptor Request function and response processing
ZDO_USERDESC_REQUEST	Enable User Descriptor Request function and response processing
ZDO_USERDESCSET_REQUEST	Enable User Descriptor Set Request function and response processing
ZDO_ENDDEVICEBIND_REQUEST	Enable End Device Bind Request function and response processing
ZDO_BIND_UNBIND_REQUEST	Enable Bind and Unbind Request function and response processing
ZDO_SERVERDISC_REQUEST	Enable Server Discovery Request function and response processing
ZDO_MGMT_NWKDISC_REQUEST	Enable Mgmt Nwk Discovery Request function and response processing
ZDO_MGMT_LQI_REQUEST	Enable Mgmt LQI Request function and response processing
ZDO_MGMT_RTG_REQUEST	Enable Mgmt Routing Table Request function and response processing
ZDO_MGMT_BIND_REQUEST	Enable Mgmt Binding Table Request function and response processing
ZDO_MGMT_LEAVE_REQUEST	Enable Mgmt Leave Request function and response processing
ZDO_MGMT_JOINDIRECT_REQUEST	Enable Mgmt Join Direct Request function and response processing
ZDO_MGMT_PERMIT_JOIN_REQUEST	Enable device to respond to Mgmt Permit Join Request function
ZDO_USERDESC_RESPONSE	Enable device to respond to User Descriptor Request function
ZDO_USERDESCSET_RESPONSE	Enable device to respond to User Descriptor Set Request function
ZDO_SERVERDISC_RESPONSE	Enable device to respond to Server Discovery Request function
ZDO_MGMT_NWKDISC_RESPONSE	Enable device to respond to Mgmt Network Discovery Request function
ZDO_MGMT_LQI_RESPONSE	Enable device to respond to Mgmt LQI Request function
ZDO_MGMT_RTG_RESPONSE	Enable device to respond to Mgmt Routing Table Request function
ZDO_MGMT_BIND_RESPONSE	Enable device to respond to Mgmt Binding Table Request function
ZDO_MGMT_LEAVE_RESPONSE	Enable device to respond to Mgmt Leave Request function
ZDO_MGMT_JOINDIRECT_RESPONSE	Enable device to respond to Mgmt Join Direct Request function
ZDO_MGMT_PERMIT_JOIN_RESPONSE	Enable device to respond to Mgmt Permit Join Request function
ZDO_ENDDEVICE_ANNCE	Enable device to respond to End Device Annce Message function
ZDO_NV_SAVE_RFDs	Default define to TRUE in ZDApp.c and only applicable if NV_RESTORE is defined. If NV_RESTORE is defined and ZDO_NV_SAVE_RFDs is defined to FALSE, then RFD joins will not trigger a call to NLME_UpdateNV() and the delay time between receiving a trigger event and actually invoking NLME_UpdateNV() is extended to the OSAL timer maximum of 65 seconds (see ZDAPP_UPDATE_NWK_NV_TIME). This compile option is intended to be used to greatly extend the life of the NV pages of the RFD's in a network with mobile or purged RFD's. When this flag is defined to FALSE, any RFD children that exist at the time an FFD is reset will not be restored and the FFD can re-issue their network addresses to other joining RFD's.
ZDAPP_UPDATE_NWK_NV_TIME	Default define to 700 msecs and only applicable if NV_RESTORE is defined. The delay time between receiving a network save state trigger event and actually invoking NLME_UpdateNV().

	The longer this delay is, the longer the life of the NV pages since this data is very large and in a busy network (especially one with mobile RFD's) the frequency of trigger events could be high.