TEXAS INSTRUMENTS

Z-Stack 3.0 Developer's Guide

Texas Instruments, Inc.
San Diego, California USA

Revision	Description	Date
1.0	Initial release	12/13/2006
1.1	Added section on ZDO Message Request	09/29/2007
1.2	Updates for ZigBee 2007 and ZigBee PRO features	02/24/2008
1.3	Updated location of `zgPreConfigKeys`	01/06/2009
1.4	Updated for 2.2.0 Release	03/30/2009
1.5	Replaced references to *ZDNwkManager* with *ZDNwkMgr*	04/14/2009
1.6	Updated section 4.1.1.1	08/03/2009
1.7	Updated section 10.5 for multiple preconfigured trust center link keys	01/15/2010
1.8	Updated section 4 Updated section 9.6.3. NV range for application use Fixed misspelling of `zgPreConfigKeys` variable Added section 10.6 on Security key data management Added section 13 on ZMAC LQI Adjustment	08/11/2010
1.9	Added Extended PAN IDs section	11/20/2010
1.10	Editorial changes to sections 2.1 and 2.1.1	03/23/2011
1.11	Added section 5.6 for Router Off-Network Association Cleanup	03/14/2012
1.12	Updated reference to ZigBee Specification (053474r20) Added description of `zgNwkLeaveRequestAllowed` variable to section 9.4 Added section 9.11 to pre-commission a device with a network address Updated section 10.6 with Unique and Global Link Key Type configuration Added section 10.8 to describe backwards interoperability Clean up the documents by removing references to specific Z-Stack applications Editorial clean up of the entire document. Added section 14: Heap Memory Management. Added section 15: Compile Options.	12/31/2012
1.13	Updated Section 7. Portable Devices to include a Rejoin State Diagram. Also made a few edits.	2/19/2015
1.14	Update the “Z-Stack Developers guide” document to reflect Z3.0 changes and merge from “Z-Stack Home Developer's Guide” and “Touchlink Developer's Guide” to this document.	10/18/2016

TABLE OF CONTENTS

List of Figures

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-2015 compliant stack for the ZigBee and ZigBee PRO stack profiles. Here is also explained the added features of the Z3.0 and how those can be set to be compatible with Z3.0 or legacy devices.

1.3 Definitions, Abbreviations and Acronyms

Term	Definition
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
BDB	Base Device Behavior
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
GP	Green Power
GPD	Green Power Device
HAL	Hardware (H/W) Abstraction Layer
MSG	Message
MT	Z-Stack's Monitor and Test Layer
NHLE	Next Higher Layer Entity
NIB	Network Information Base
NWK	Network
OSAL	Z-Stack's Operating System Abstraction Layer
OTA	Over-the-air
PAN	Personal Area Network
RSSI	Received Signal Strength Indication

SE	Smart Energy
Sub-Device	A self contained device functionality in a Zigbee device application endpoint.
TC	Trust Center
TCLK	Trust Center Link Key
ZCL	ZigBee Cluster Library
ZDO	ZigBee Device Object
ZHA	ZigBee Home Automation
ZC	ZigBee Coordinator
ZR	ZigBee Router
ZED	ZigBee End Device

1.4 Reference Documents

[1] Texas Instruments document SWRA195, Z-Stack API
[2] Texas Instruments document SWRA194, Z-Stack OSAL API
[3] Texas Instruments document SWRU354, Z-Stack 3.0 Sample Application User's Guide.
[4] Texas Instruments document SWRA353, Z-Stack OTA Upgrade User's Guide.
[5] ZigBee document 05-3474-21 ZigBee ZigBee Specification
[6] ZigBee document 07-5123-06 ZigBee Cluster Library Specificiation
[7] ZigBee document 13-0402-13 ZigBee Base Device Behavior
[8] ZigBee document 14-0563-16 ZigBee Green Power specification

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 device with formation capabilities (such as Coordinator or Router) 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.

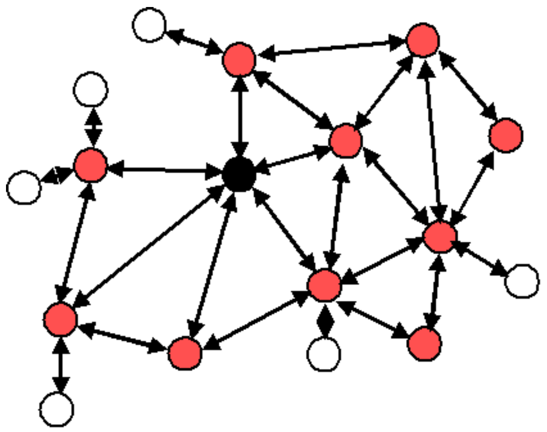


Figure 1: Example of typical ZigBee network

An example network is shown in Figure 1, with the ZigBee Coordinator (black), the Routers (red), and the End Devices (white).

2.1.1 Coordinator

A coordinator is a device with network formation capabilities, but without network joining capabilities. It means it can only create its own network, but not join existing networks. To create a 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. In Z3.0 this device creates a Centralized security network and is mandated to behave as the Trust Center of this network, which means that this device is responsible to manage the security of the network and it is the only device capable of distributing keys and allowing devices to join the network it has created.

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

The role of the coordinator is mainly related to starting the network and managing the keys, besides that, it behaves like a router device. It is important to note that network procedures related to devices joining or leaving the network must be attended by the Coordinator, hence it cannot be absent of its own network. Further details on security schemas are available in section 10.

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 end devices. In Z3.0 this device has been granted with formation capabilities that allow it to create a Distributed security network. This formation capability allows the router device to create a network that does not have a security manager. This means that once the network has been created, the router which created it does not have any special role in this network. More details are available in section 10.

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.

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

Notes:
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.

2.2 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).

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 legacy ZigBee stack profile has ID of 1, and a ZigBee PRO stack profile (which is used for Z3.0) has ID of 2. The stack profile is configured by the `STACK_PROFILE_ID` parameter in *nwk_globals.h* file. The stack profile of 3 is reserved for Green Power devices and it appears in the respective frames.

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 `AF_DataRequest()` function. The destination device to which the packet is to be sent is of type `afAddrType_t` (defined in `ZComDef.h`).

```
typedef struct
{
  union
  {
    uint16      shortAddr;
    ZLongAddr_t extAddr;
  } addr;
  afAddrMode_t  addrMode;
  byte endPoint;
} afAddrType_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 `AF.h`)

```
typedef enum
{
  afAddrNotPresent = AddrNotPresent,
  afAddr16Bit      = Addr16Bit,
  afAddr64Bit      = Addr64Bit,
  afAddrGroup      = AddrGroup,
  afAddrBroadcast  = AddrBroadcast
} afAddrMode_t;
```

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 `Addr16Bit` 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 `AddrNotPresent` 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 `AddrBroadcast` and the destination address can be set to one of the following broadcast addresses:
`NWK_BROADCAST_SHORTADDR_DEVALL (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`).
`NWK_BROADCAST_SHORTADDR_DEVRXON` (`0xFFFD`) – the message will be sent to all devices that have the receiver on when idle (`RXONWHENIDLE`). That is, all devices except sleeping devices.
`NWK_BROADCAST_SHORTADDR_DEVZCZR` (`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 `afAddrGroup` and the `addr.shortAddr` is set to the group identifier.

Before using this feature, groups must be defined in the network, see `aps_AddGroup()` in the Z-Stack API [1] 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:

```
aps_Group_t group;

// Assign yourself to group 1
group.ID = 0x0001;
group.name[0] = 6;  // First byte is string length
osal_memcpy( &(group.name[1]), "Group1", 6);
aps_AddGroup( SAMPLEAPP_ENDPOINT, &group );
```

3.4 Important Device Addresses

An application may want to know the address of its device and that of its parent. Use the following functions to get this device's address, defined in Z-Stack API [1] document:

- `NLME_GetShortAddr()` – returns this device's 16 bit network address.
- `NLME_GetExtAddr()` – returns this device's 64 bit extended address.
- Use the following functions to get this device's parent's addresses, defined in Z-Stack API [1] document. Note that the term "Coord" in these functions does not refer to the ZigBee Coordinator, but instead to the device's parent (MAC Coordinator):
- `NLME_GetCoordShortAddr()` – returns this device's parent's 16 bit short address.
- `NLME_GetCoordExtAddr()` – returns this device's parent's 64 bit extended address.

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 4 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.
- Finding and Binding commissioning process for initiator devices.

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 `ZDP_BindReq()` [defined in `ZDProfile.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 `ZDP_UnbindReq()` can be used, with the same parameters, to remove the binding record.

The target device will send back a ZigBee Device Object Bind or Unbind Response message which the ZDO code on the coordinator will parse and notify `ZDApp.c` by calling `ZDApp_ProcessMsgCBs()` 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.

For the Coordinator End Device Binding process, the coordinator registered `ZD_RegisterForZDOMsg()` to receive End Device Bind Request, Bind Response and Unbind Response ZDO messages in `ZDApp_RegisterCBs()` defined in `ZDApp.c`. When these message are received they are sent to `ZDApp_ProcessMsgCBs()`, where they are parsed and processed.

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. Meaning that the application will enter and remove binding table entries locally by calling the following binding table management functions, see Z-Stack API [1] Document – Binding Table Management section:

- `bindAddEntry()` – Add entry to binding table
- `bindRemoveEntry()` – Remove entry from binding table
- `bindRemoveClusterIdFromList()` – Remove a cluster ID from an existing binding table entry
- `bindAddClusterIdToList()` – Add a cluster ID to an existing binding table entry
- `bindRemoveDev()` – Remove all entries with an address reference
- `bindRemoveSrcDev()` – Remove all entries with a referenced source address
- `bindUpdateAddr ()` – Update entries to another address
- `bindFindExisting ()` – Find a binding table entry
- `bindIsClusterIDinList()` – Check for an existing cluster ID in a table entry
- `bindNumBoundTo()` – Number of entries with the same address (source or destination)
- `bindNumOfEntries()` – Number of table entries
- `bindCapacity()` – Maximum entries allowed
- `BindWriteNV()` – Update table in NV.

4.1.3 Finding and binding

Base Device Behavior has defined a commissioning method called Finding and Binding, which is a process that relies on the usage of the Identify cluster and ZDO messages to allow the commissioned device to find devices with matching application clusters. This mechanism is usually triggered by the user to specify which devices need to "Find and Bind" each other so these pairs of devices can communicate more effectively. Refer to 15.7.2 for further details on this commissioning method.

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.

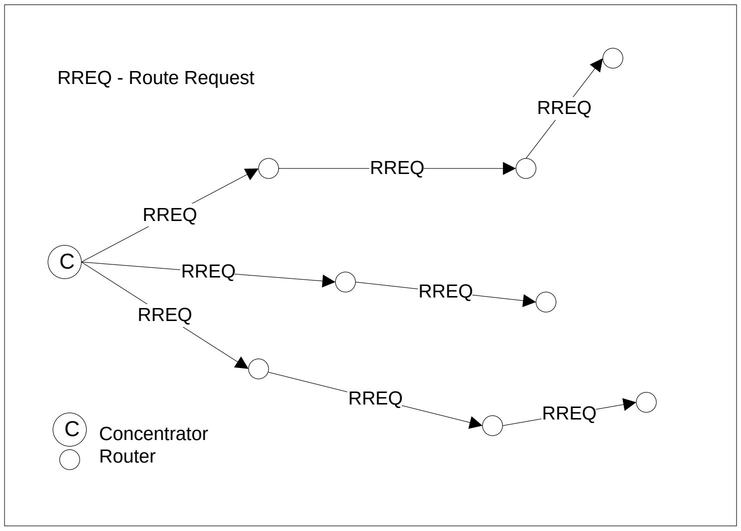


Figure 2: Many-to-one route discovery illustration

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 API [1] documentation for detailed usage about this API.

```
  ZStatus_t NLME_RouteDiscoveryRequest( uint16 DstAddress,
                                        byte options, uint8 radius )
```

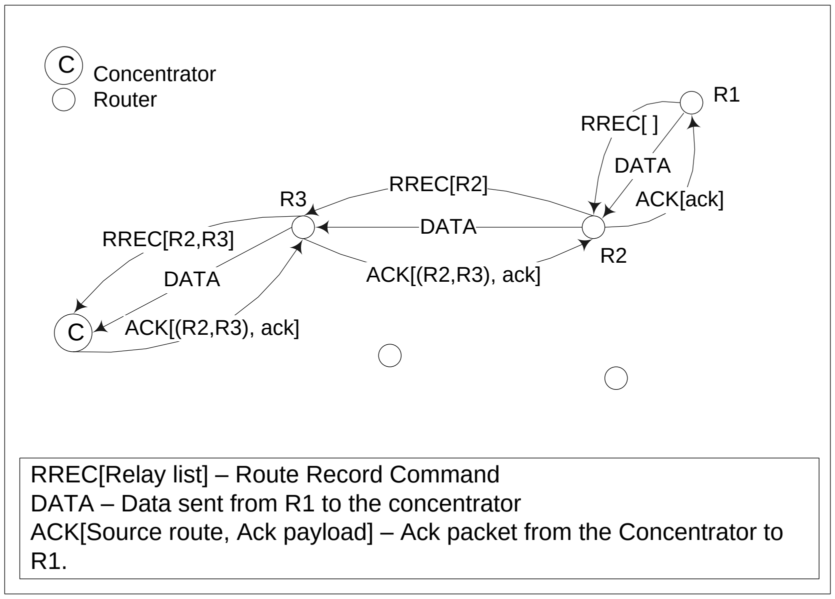
The option field is a bitmask to specify options for the route request. It can have the following values:

Value	Description
0x00	Unicast route discovery
0x01	Many-to-one route discovery with route cache (the concentrator does not have memory constraints).
0x03	Many-to-one route discovery with no route cache (the concentrator has memory constraints)

When the option field has value 0x01 or 0x03, the `DstAddress` field will be overwritten with the many-to-one destination address 0xFFFC. Therefore, user can pass any value to `DstAddress` 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 3. 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.



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 3: 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. When the concentrator receives the route failure, the application will decide whether or not to re-issue a many-to-one route request.

When the concentrator receives network status command indicating many-to-one route failure, it passes the indication to the ZDO layer and the following ZDO callback function in `ZDApp.c` is called:

```
  void ZDO_ManytoOneFailureIndicationCB()
```

By default, this function will redo a many-to-one route discovery to recover the routes. You can modify this function if you want a more complicated process other than the default.

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`)

5.6 Router Off-Network Association Cleanup

In case a ZigBee Router gets off network for a long period of time, its children will try to join an alternative parent. When the router is back online, the children will still appear in its child table, preventing proper routing of egress traffic to them.

In order to avoid this, it is recommended that routers prone to get off and on the network will have `zgRouterOffAssocCleanup` flag set to `TRUE` (mapped to NV item: `ZCD_NV_ROUTER_OFF_ASSOC_CLEANUP`):

```
  uint8 cleanupChildTable = TRUE;
  zgSetItem( ZCD_NV_ROUTER_OFF_ASSOC_CLEANUP, sizeof(cleanupChildTable),
             &cleanupChildTable );
```

When enabled, deprecated end device entries will be removed from the child table if traffic received from them was routed by another parent.

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.

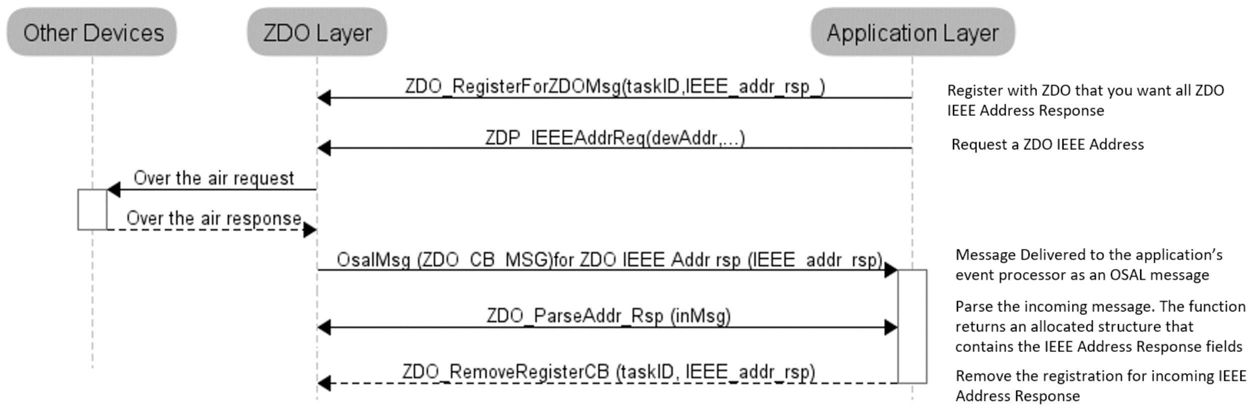

Other Devices
ZDO Layer
Application Layer
ZDO_RegisterForZDOMsg(taskID,IEEE_addr_rsp_)
Register with ZDO that you want all ZDO IEEE Address Response
ZDP_IEEEAddrReq(devAddr,...)
Request a ZDO IEEE Address
Over the air request
Over the air response
OsalMsg (ZDO_CB_MSG)for ZDO IEEE Addr rsp (IEEE_addr_rsp)
Message Delivered to the application's event processor as an OSAL message
ZDO_ParseAddr_Rsp (inMsg)
Parse the incoming message. The function returns an allocated structure that contains the IEEE Address Response fields
ZDO_RemoveRegisterCB (taskID, IEEE_addr_rsp)
Remove the registration for incoming IEEE Address Response

Figure 4: 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.

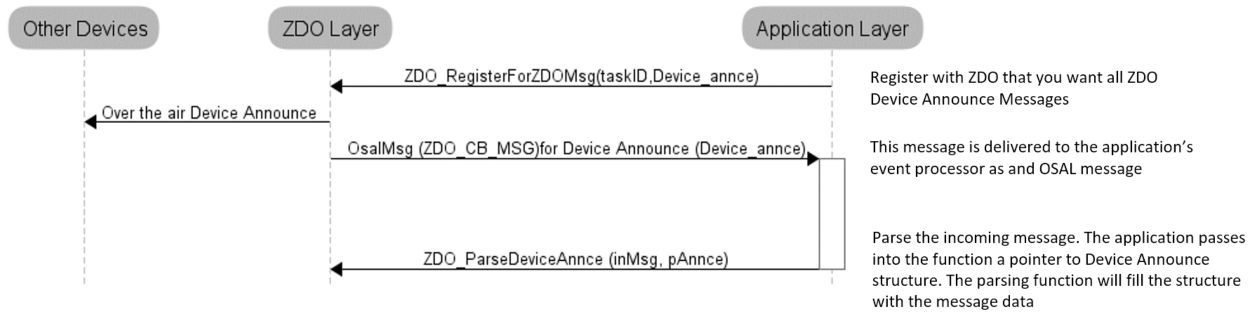

Other Devices
ZDO Layer
Application Layer
ZDO_RegisterForZDOMsg(taskID,Device_annce)
Register with ZDO that you want all ZDO Device Announce Messages
Over the air Device Announce
OsalMsg (ZDO_CB_MSG)for Device Announce (Device_annce)
This message is delivered to the application's event processor as and OSAL message
ZDO_ParseDeviceAnnce (inMsg, pAnnce)
Parse the incoming message. The application passes into the function a pointer to Device Announce structure. The parsing function will fill the structure with the message data

Figure 5: ZDO Device Announce delivered to an application

7. Portable Devices

An 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 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 notify the application that it has lost its parent through the BDB interface (see section 15.3), then the application is responsible for managing the rejoining of the device by using the BDB API `bdb_ZedAttemptRecoverNwk()`(Described in [1]), which will trigger the process of scanning the channel in which this device was commissioned, in order to search another suitable parent device. It is recommended that as soon as an end device loses its parent, it should try to recover. If recovery fails, the device should try once again after a short delay, and if it still fails, it should retry periodically with a larger waiting period. This practice allows for better power usage on the end device and does not interfere with other networks that may be on the same channel.

In secure networks, 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 such end devices are re-established automatically.

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 `options` field of the call to `AF_DataRequest()`. The options field is a bit map of options, so OR in `AF_ACK_REQUEST` to enable APS ACK for the message that you are sending. 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 Z3.0 device has a primary channel mask configuration (`BDB_DEFAULT_PRIMARY_CHANNEL_SET`) and a secondary channel mask configuration (`BDB_DEFAULT_SECONDARY_CHANNEL_SET`). For devices with formation capabilities that were instructed to create a network, these channels masks are used when scanning for a channel with the least amount of noise to create the network on. For devices with joining capabilities that were instructed to join a network, these channel masks are used when scanning for existing networks to join. The device will try first with all the channels defined in the primary channel mask and then if the process is not successful (the network were not created or no network to join was found) the secondary channel mask is used. These two channel masks can be configured by the application as needed. A value of 0 in one of these masks will disable the respective channel scanning phase (primary or secondary). The primary channel mask is defined by default to be equal to `DEFAULT_CHANLIST` (in `f8wConfig.cfg`), while the secondary channel mask is defined as all the other channels (i.e. `DEFAULT_CHANLIST ^ 0x07FFF800`). Section 15 provides more details on the commissioning methods.

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. It can also be used to pre-set the PAN ID of a new network to be created by a coordinator or a router. The `ZDO_CONFIG_PAN_ID` parameter in `f8wConfig.cfg` can be set to a value (between 1 and 0xFFFE). A coordinator or a network-forming router will use this value as the PAN ID of the network when instructed to create a network. A joining router or end device will only join a network that has a PAN ID that matches the value of this parameter. To turn this feature off, set the parameter to a value of 0xFFFF. In this case, a newly created network will have a random PAN ID, and a joining device will be able to join any network regardless of its PAN ID.

The network discovery process is managed by the Network Steering commissioning process, which is explained 15.5. It allows filtering of the discovered networks, by registering a callback with the function `bdb_RegisterForFilterNwkDescCB()`, to which the application receives a list of network descriptors of the networks found on each scan attempt (primary channels first and then another call for secondary channel if performed). The application may skip attempting to join specific networks by freeing the network descriptors using `bdb_nwkDescFree()`. For details on these API refer to [1].

For further control of the joining procedure, the `ZDO_NetworkDiscoveryConfirmCB` function in the `ZDApp.c` should be modified. `ZDO_NetworkDiscoveryConfirmCB()` is called when the network layer has finished with the Network Discovery process, started by calling `NLME_NetworkDiscoveryRequest()`, detailed in the Z-Stack API [1] document.

9.3 Maximum payload size

The maximum payload size for an application is based on several factors. The MAC layer provides a constant payload length of 116 (can be changed in `f8wConfig.cfg – MAC_MAX_FRAME_SIZE`). The NWK layer requires a fixed header size, one size with security and one without security. The APS layer has a required, but variable, header size based on a variety of settings, including the ZigBee Protocol Version, APS frame control settings, etc. Ultimately, the user does not have to calculate the maximum payload size using the aforementioned factors. The AF module provides an API that allows the user to query the stack for the maximum payload size, or the maximum transport unit (MTU). The user can call the function, `afDataReqMTU()` (see `AF.h`) which will return the MTU, or maximum payload size.

```
typedef struct
{
  uint8              kvp;
  APSDE_DataReqMTU_t aps;
} afDataReqMTU_t;

uint8 afDataReqMTU( afDataReqMTU_t* fields )
```

Currently the only field that should be set in the `afDataReqMTU_t` structure is `kvp`, which indicates whether KVP is being used and this field should be set to FALSE. The `aps` field is reserved for future use.

9.4 Leave Network

The ZDO Management implements the function `ZDO_ProcessMgmtLeaveReq()`, which provides access to the "NLME-LEAVE.request" primitive. "NLME-LEAVE.request" allows a device to remove itself or remove a remote device from the network. The `ZDO_ProcessMgmtLeaveReq()` removes the device based on the provided IEEE address. When a device removes itself, it will wait for LEAVE_RESET_DELAY (5 seconds by default) and then reset. When a device removes a child device, it also removes 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 R21 of the ZigBee PRO specification, 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`. Processing of these commands depending on the logical device type has also changed: Coordinators do not process leave commands, Router devices process leave commands from *any* device in the network (if allowed as mentioned above), and end devices only process leave commands from their parent device.
In the base device behavior specification is also stated that if any device receives a valid leave request with rejoin set to FALSE (meaning that this device shall not rejoin the network), then that device is forced to perform a Factory New reset. In this case, Z-Stack clears all the ZigBee persistent data, while it is up to the application to clear the relevant application data from Nv.

9.5 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.

9.6 Non-Volatile Memory Items

9.6.1 Global Configuration Non-Volatile Memory

Global device configuration items are stored in `ZGlobal.c`. This includes items such as PAN ID, key information, network settings, etc.. The default values for most of these items are specified in `f8wConfig.cfg`. These items are loaded to RAM at startup for quick accessed during Z-Stack operation. To initialize the non-volatile memory area to store these items, the compile flag NV_INIT must be enabled in your project (it is enabled by default in the sample applications).

9.6.2 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.

This feature is enabled by default by the inclusion of the `NV_RESTORE` compile option. Note that this feature must be always enabled in a real ZigBee network. The ability to disable 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, but it is the BDB layer which will define when to retrieve this information or when to clear and start as "factory new" device. 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. This is also used for security to store frame counters and keys.
If the device is not meant to be set to its factory new state, the device will then use this information to restore the device in the network if the device is reset by any mean.
Upon initializing, the BDB layer will check the attribute `bdbNodeIsOnANetwork` to know if this device was commissioned to a network. If it was commissioned to a network and it was also instructed to resume operations in the same network then the BDB layer will call `ZDOInitDeviceEx()`, which will handle the resume operation according to the state and the logical device type.

9.6.3 Application Non-Volatile Memory

In general, a device must have non-volatile memory enabled to be certified, because it must remember its network configuration. In addition to the stack 'internal' data, the NVM can also be used to store application data.

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

Once the item ID is selected, the item must be initialized using `osal_nv_item_init()`. This function receives the item ID, length of the variable to be stored, and an initial value.

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

To read the item from NVM, use `osal_nv_read()`. This function receives the item ID, the index offset into an item, length of data to read, and the variable to read the data to.

These functions can be found in the *OSAL_Nv.c* file.

9.7 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`. 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.

9.8 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 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 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.

9.9 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.

To turn on the APS Fragmentation feature in your Z-Stack project include the ZIGBEE_FRAGMENTATION compile flag. By default, all projects where ZIGBEEPRO is defined include fragmentation and there is no need to add the ZIGBEE_FRAGMENTATION compile flag. All applications using fragmentation will include the APS Fragmentation task APSF_Init() and APSF_ProcessEvent(). If you have an existing application, make sure the code in the OSAL_xxx.c of your application has included the header file:

```
#if defined ( ZIGBEE_FRAGMENTATION )
  #include "aps_frag.h"
#endif
```

And in tasksArr[] there is an entry for APSF_ProcessEvent(), like in the example below:

```
const pTaskEventHandlerFn tasksArr[] = {
  macEventLoop,
  nwk_event_loop,
#if (ZG_BUILD_RTR_TYPE)
  gp_event_loop,
#endif
  Hal_ProcessEvent,
#if defined( MT_TASK )
  MT_ProcessEvent,
#endif
  APS_event_loop,
#if defined ( ZIGBEE_FRAGMENTATION )
  APSF_ProcessEvent,
#endif
  ZDApp_event_loop,
#if defined ( ZIGBEE_FREQ_AGILITY ) || defined ( ZIGBEE_PANID_CONFLICT )
```

```
  ZDNwkMgr_event_loop,
#endif
  zcl_event_loop,
  bdb_event_loop,
  xxx_ProcessEvent          /* Where xxx is your application's name */
};
```

And `osalInitTasks()` function calls `APSF_Init()`, like in the code below;

```
void osalInitTasks( void )
{
  uint8 taskID = 0;

  tasksEvents = (uint16 *)osal_mem_alloc( sizeof( uint16 ) * tasksCnt);
  osal_memset( tasksEvents, 0, (sizeof( uint16 ) * tasksCnt));

  macTaskInit( taskID++ );
  nwk_init( taskID++ );
#if (ZG_BUILD_RTR_TYPE)
  gp_Init( taskID++ );
#endif
  Hal_Init( taskID++ );
#if defined( MT_TASK )
  MT_TaskInit( taskID++ );
#endif
  APS_Init( taskID++ );
#if defined ( ZIGBEE_FRAGMENTATION )
  APSF_Init( taskID++ );
#endif
  ZDApp_Init( taskID++ );
#if defined ( ZIGBEE_FREQ_AGILITY ) || defined ( ZIGBEE_PANID_CONFLICT )
  ZDNwkMgr_Init( taskID++ );
#endif
  zcl_Init( taskID++ );
  bdb_Init( taskID++ );
  xxx_Init( taskID );       /* Where xxx is your application's name */
}
```

When APS Fragmentation is turned on, sending a data request with a payload larger than a normal data request payload will automatically trigger fragmentation.

Fragmentation parameters are in the structure `afAPSF_Config_t`, which is part of the Endpoint Descriptor list `epList_t` defined in `AF.h`, default values for these parameters are used when calling `afRegister()`, to register the Application's Endpoint Descriptor, which in turn calls `afRegisterExtended()`, the default values `APSF_DEFAULT_WINDOW_SIZE` and `APSF_DEFAULT_INTERFRAME_DELAY` are defined in `ZGlobals.h`:

- `APSF_DEFAULT_WINDOW_SIZE` - The size of a Tx window when using fragmentation. This is the number of fragments that are sent before an APS Fragmentation ACK is expected. So, if the message is broken up into 10 fragments and the max window size is 5, then an ACK will be sent by the receiving device after 5 fragments are received. If one packet of the window size isn't received, the ACK is not sent and all the packets (within that window) are resent.
- `APSF_DEFAULT_INTERFRAME_DELAY` – The delay between fragments within a window. This is used by the sending device.

These values can be read and set by the application by calling `afAPSF_ConfigGet()` and `afAPSF_ConfigSet()` 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.

9.9.1 Quick Reference

Compile flag to activate the feature	`ZIGBEE_FRAGMENTATION`
Maximum fragments in a window default value	`APSF_DEFAULT_WINDOW_SIZE` (defined in `ZGlobals.h`)
Interframe delay default value	`APSF_DEFAULT_INTERFRAME_DELAY` (defined in `ZGlobals.h`)
Application/Profile maximum buffer size	`MAX_TRANSFER_SIZE` (defined in `ZDConfig.h`)

9.10 Extended PAN IDs

There are two Extended PAN IDs used in the Z-Stack:

- `zgApsUseExtendedPANID`: This is the 64-bit PAN identifier of the network to join or form. This corresponds to the `ZCD_NV_APS_USE_EXT_PANID` NV item.
- `zgExtendedPANID`: This is the 64-bit extended PAN ID of the network to which the device is joined. If it has a value of 0x0000000000000000, then the device is not connected to a network. This corresponds to the `ZCD_NV_EXTENDED_PAN_ID` NV item.

If the device has formation capabilities and is instructed to form a network, then it will form a network using `zgApsUseExtendedPANID` if `zgApsUseExtendedPANID` has a non-zero value. If `zgApsUseExtendedPANID` is 0x0000000000000000, then the device will use its 64-bit Extended Address to form the network.

9.11 Rejoining with Pre-Commissioned Network parameters

In previous ZigBee stacks, it was possible for a rejoining device to use a pre-configured network address. As of today, the Base Device Behavior specification has not addressed this topic (whether this is allowed or not). TI encourages the use of the Base Device Behavior commissioning methods described in the section 15 for rejoining the network.

.

9.12 Child management

R21 (revision 21 of the ZigBee specification, AKA ZigBee 2015), has introduced a child management feature that is meant to allow end device mobility while allowing parent devices to purge its tables from end devices which are no longer their children. When an end device joins/rejoins it will send an EndDeviceTimeout nwk command, which tells its parent device a time period after which it can remove it from its association table, if the device has not sent a keep-alive message. The parent device will answer this network command with a response stating which methods it supports for receiving the keep-alive messages. At the moment of this release only one keep-alive method is specified, which uses the standard MAC polling. If a legacy device joins an R21 or later parent device, the parent will assign a default timeout to expire this device if this legacy device fails to poll in a timely manner. Additionally if a parent device is polled by an end device which is not its child (due to being expired or not being its child at all), then the parent device must request this end device to leave the network with rejoin set to TRUE, so this device can rejoin the network and find a new parent (which could be the same router or another one).

9.12.1 Configuring child management for parent device

A default end device timeout (for both legacy and R21 end devices) can be defined in the parent device by modifying `NWK_END_DEV_TIMEOUT_DEFAULT`. This timeout will be overwritten by joining devices if they state their own timeout using the EndDeviceTimeout command.

Parent devices must keep track of devices that should be sent a leave request, due to being expired or end devices polling this parent due to unknown reasons. To do this, parent device must queue a leave request in the mac layer. The number of devices that can be keep track of at the same time is defined by `MAX_NOT_MYCHILD_DEVICES`. These devices will be tracked for a period of time defined by `NWK_END_DEVICE_LEAVE_TIMEOUT`. All these parameters are defined in *ZGlobals.h*.

9.12.2 Configuring child management for child devices

The timeout that the child device will indicate to the parent device is defined by `END_DEV_TIMEOUT_VALUE` and it is suggested to be at least 3 times greater than the MAC polling time to avoid being expired if there is interference when the end device is polling.

9.12.3 Parent Annce

The child management functionality includes the usage of Parent Annce ZDO message which is broadcasted by parent devices and contains the 64-bit IEEE address of all end devices in the parent's association table. This message is send only when forming a network or being reset, after 10 seconds plus a random jitter of up to 10 seconds. If this message is receive by a different parent device, it will check if any of the reported children is also listed in its association table. If any match is found then this parent device will reply to the originator of the message, indicating which children are no longer its children. The usage of this message can be illustrated with the following example:

1) Parent device 'A' has a child device 'c'.
2) Parent device 'A' is power cycled.
3) Child device 'c' finds parent device 'B' and joins it.
4) When parent device 'A' restores its network parameters, it starts a timer to send parent annce (of 10 seconds plus random jitter of up to 10 seconds.)
5) After the timeout parent 'A' device broadcasts parent annce containing IEEE address of child 'c'.
6) Parent device 'B' finds a match with its children and responds with a parent annce response containing the IEEE address of child 'c'.
7) Parent device removes child 'c' from its table.

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 specification defines two types of networks, based on the security schemas that those networks use: Centralized security network and Distributed security network.
By default, networks are closed for new devices. In both types of networks, the network can only be opened for a maximum of 254 seconds at a time, after which the network will be closed for joining. Z3.0 networks cannot remain open indefinitely. The duration for which devices may attempt to join a network is reflected in the beacon packets sent by any existing networks in response to a joining devices beacon requests.

ZigBee offers the following security features:

- Infrastructure security
- Network access control
- Application data security (only for centralized security networks)

10.2 Configuration

To use network layer security, all device images must be built with the preprocessor flag `SECURE` set equal to 1. This can be found in the *f8wConfig.cfg* file and is enabled by default in all projects, as it is mandatory in Z3.0

The default key for network layer encryption (`defaultKey` defined in *nwk_globals.c*) can be either preconfigured on all joining/network-forming devices or it can be distributed to each joining device over-the-air as they join the network. This is chosen via the `zgPreConfigKeys` option in *ZGlobals.c*. 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 device forming the network. This `defaultKey` is initialized with the macro definition `DEFAULT_KEY` in *f8wConfig.cfg*. If this key is set to 0 upon initialization, then a random key will be generated. In Z3.0 this key is transmitted over-the-air to joining devices using APS layer encryption.

10.3 Centralized security network

This network type is formed by coordinator devices, in which the coordinator assumes the role of TC. In this type of network only the TC can deliver the network key to joining and allow them to be part of the network. The coordinator can configure different sets of TC policies that allow control of the security level of the network, these policies will be presented in section 10.3.1. When a device performs an association directly to the TC, the TC will evaluate the TC policies and validate if the device is allowed to join the network or not. When a device joins through a router device, the parent device notifies the TC via an APS Update Device command, and then the joining device will go through the same TC policy validations. If a device passes the validations, the TC will deliver the network key to the joining device through either a direct APS Transport Key command or an APS Tunnel: Transport Key command, depending on the devices joining topology. If the joining device does not pass the TC policy validations, it will be kicked out of the network with a network leave command.
It is also important to note that if the TC is not available (power cycled or not in the network), new devices will not be able to join the network since no other device is allowed to deliver the network key or validate TC policies.

10.3.1 Trust center policies

10.3.1.1 zgAllowRemoteTCPolicyChange

If this policy is set to `TRUE,` other devices in the network may modify the permit joining policy of the trust center, which could allow other devices to join the network. If set to `FALSE`, remote devices will not be able to change the permit joining policy on the coordinator, which will cause the TC to not deliver the network key and kick out any devices attempting to join the network through an intermediate router which may have locally enabled permit join.

10.3.1.2 bdbJoinUsesInstallCodeKey

If `bdbJoinUsesInstallCodeKey` is set to `TRUE`, then the network key will be delivered only to those joining devices that do have an install code associated. If `bdbJoinUsesInstallCodeKey` is set to `FALSE`, joining devices may use install codes. The usage of install codes is described in section 10.5.2.

10.3.1.3 bdbTrustCenterRequireKeyExchange

If this policy is set to `TRUE` (set to this value by default in *bdb_interface.h*) all the joining devices are mandated to perform the TCLK exchange procedure. Devices that do not perform this procedure will be kicked out of the network after `bdbTrustCenterNodeJoinTimeout` seconds (15 by default). If this policy set to `FALSE,` joining devices will not be required to perform a TCLK update, but they will be allowed to do so. The TCLK exchange procedure is described in section 10.6.1.
It is important to note that legacy devices (implementing R20 or before) will not be able to perform the TCLK exchange process, so if this policy is set to `TRUE`, legacy devices will not be able to join this network.

10.3.2 Key Updates

The Trust Center can update the common Network key at its discretion. 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 not a unicast address. Once the Trust Center calls `ZDSecMgrSwitchNwkKey()`, with the `dstAddr` of the device or devices, all destination devices will use their alternate key.

In R21 revision of the ZigBee specification, the network frame counter is mandated to be persistent across factory new resets, but it can be reset to 0 under the following condition: if the network frame counter is larger than half of its max value (0x8000000) prior to performing a network key update, performing the update will reset the frame counter to 0.

10.4 Distributed security network

This network type can be formed by network-forming router devices. In this network topology, all the nodes have the ability to open the network for joining and any router device can deliver the network key to a joining device. The network key will be encrypted at APS layer with a Default Distributed Global key, in section 10.5.3. This network key will be delivered via an APS Transport Key Command in which the TC address will be set to 0xFFFFFFFFFFFFFFFF, which tells the joining device it is joining a distributed security network. The application can consult the value of `AIB_apsTrustCenterAddress` to see if it has joined a distributed network.
It is important to note that after a distributed network is formed, the network key cannot be updated because there is no defined method of securely distributing a network key in a network with this topology.

10.5 Link Key types

Each node must support a way to use the following link key types:

1. The default global Trust Center link key (Used by Z-Stack automatically).
2. The distributed security global link key (Used by Z-Stack automatically).
3. An install code derived preconfigured link key (Refer to Z-Stack API to enable set this [1]).
4. The touchlink preconfigured link key (if touchlink enabled).

10.5.1 Default global Trust Center link key

All devices share a default global Trust Center Link Key. This is an APS layer key and is the first key to be used when joining a network, if no other link key is specified. This key cannot be modified if interoperability with other Z3.0 devices is desired.

Default global Trust Center link key (0:15)	=	0x5a 0x69 0x67 0x42 0x65 0x65 0x41 0x6c 0x6c 0x69 0x61 0x6e 0x63 0x65 0x30 0x39

10.5.2 Install Code Derived Trust Center Link Key

An Install Code is a sequence of 16 bytes followed by 2 bytes of CRC. A complete 18 bytes sequence is needed to generate a unique TCLK. The usage of install codes defined in Z3.0 was added to allow a generalized out-of-band key delivery method for network commissioning. It works as follows:

1. TC gets the install code and the 64-bit IEEE address of the device that will use this install code to join, via any user interface (serial, display and switches, etc.). The install code must be physically provided with the joining device.
2. TC validates the CRC of the install code introduced. If this is valid then a TCLK entry is added into the TC with the derived key and the address of the corresponding device.
3. The joining device is instructed to use its install code to generate the corresponding TCLK.
4. The network is open by any means.
5. The joining device performs association and the Trust Center delivers the network key encrypted in APS layer with the install code derived key.
6. After this, the joining device must perform the update of its TCLK as BDB specification requires.

For further details on how to generate the install codes, see the Base Device Specification [7]. This is supported only by R21 or later revisions, so to allow backwards compatibility the application must have a way to attempt joining networks without the usage of Install Codes.

10.5.3 Distributed security global link key

When a device joins to a distributed security network (no trust center), the network key will be delivered encrypted in APS layer with the Distributed security global link key by the parent router device. This key cannot be modified if interoperability with other Z3.0 devices is desired.

Distributed Trust Center link key (0:15)	=	0xd0 0xd1 0xd2 0xd3 0xd4 0xd5 0xd6 0xd7 0xd8 0xd9 0xda 0xdb 0xdc 0xdd 0xde 0xdf

10.5.4 Touchlink preconfigured Link Key

This key is used for development when a device is used to join a network using the Touchlink commissioning procedure.

Touchlink preconfigured link key (0:15)	=	0xc0 0xc1 0xc2 0xc3 0xc4 0xc5 0xc6 0xc7 0xc8 0xc9 0xca 0xcb 0xcc 0xcd 0xce 0xcf

10.6 Unsecure join to a network

Base Device Behavior has defined the procedure in which a device has to commission itself into a network from a factory new state; this process involves how the joining device performs the discovery of the available networks in multiple channels and how it fallbacks to discover additional networks in the remaining channels, this is described in section 15.5. Once the device has selected a suitable network, the joining device will perform an unsecure association (this term refers to a joining device does not have the network key) and the joining device will wait to receive the network key, from which the joining device will determinate if it has joined a Centralized security network or a Distributed security network. These networks use different keys to encrypt the APS Transport command containing the network key, as defined above. The specific secure procedures to join these types of secure networks will be explained in the following subsections.

10.6.1 Unsecure join to a centralized network

Once the transport key is received by the joining device, it will proceed to check the source address of this transport key command. In this case the 64-bit IEEE address will be different from 0's and FF's, since the TC exists in this network. The following steps describe the unsecure joining process to a Centralized network. The joining process into a Z3.0 Centralized network directly to the TC is illustrated in Figure 16.

1. Joining device sends association request.
2. Parent device sends association response.
3. Trust Center delivers the network key in a Transport key command. This transport key command is APS encrypted either with Default Global Centralized Key (10.5.1) or an Install Code derived key (10.5.2)
4. Joining device is able to get the network key from the encrypted Transport Key command and announces itself with a ZDO device announce command.
5. The joining device then queries the ZDO Node Descriptor from the trust center.
6. The joining device parses the Node Descriptor to look at the stack version revision (this field has been added by R21 version of ZigBee specification [5].
   a. If the stack version supported by the TC is not present (0x00), this means it supports a version from before to R21, so the joining process will finish at this step.
   b. If the TC of the joined network is R21 or later, the joining device must update its APS Key. This is done by the joining device by sending an APS Request Key command.
7. The TC will deliver the Unique Trust Center link key with an APS Transport Key command.
8. The joining device will update its key from *Defautl* status or *Provisional* status if install code was used, to *Unverified*, after which the key must be verified. To verify the key, the joining device will send an APS Verify Key command to the TC containing the *Unique* key hashed (to avoid sending the key in plain text).
9. The TC hashed the key associated to this device and compares against the hash received. If those are the same, it will send an APS Confirm Key command with status *Success*, after which the TCLK exchange procedure is finish for the joining device.

The joining device will attempt up to `BDBC_REC_SAME_NETWORK_RETRY_ATTEMPS` each step from steps 1 to 4, upon failure in any of these steps; the device will retry the next suitable network in the network descriptor list. In the same way, the steps 5 to 8 are attempted up to `BDB_DEFAULT_TC_LINK_KEY_EXCHANGE_ATTEMPS_MAX` times each step, upon failure in any of these steps; the device will perform a Factory New reset, to erase the network parameters and keys obtained at the failing step. The application will receive a notification on these according to section 15.1.

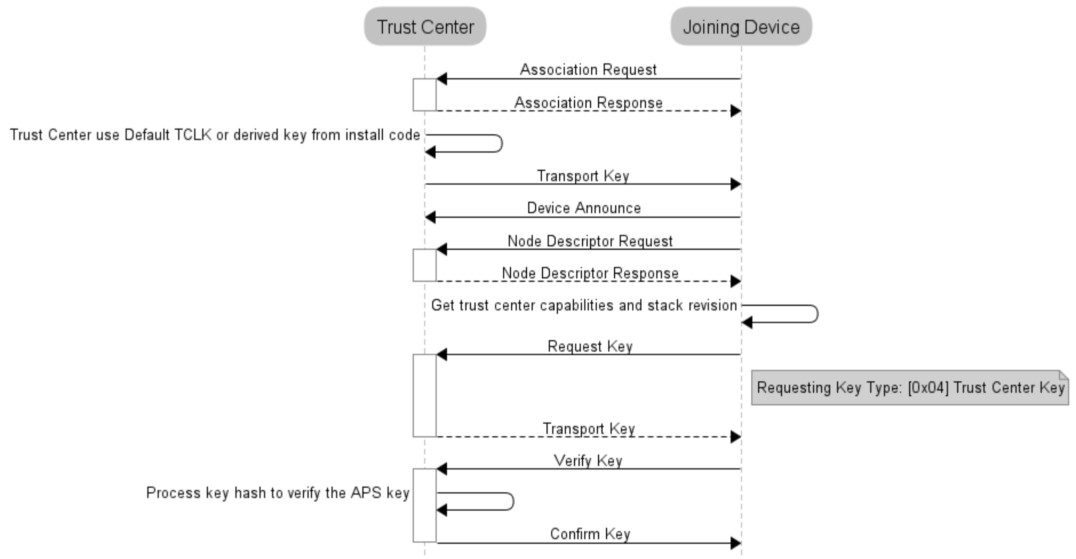

Trust Center
Joining Device
Association Request
Association Response
Trust Center use Default TCLK or derived key from install code
Transport Key
Device Announce
Node Descriptor Request
Node Descriptor Response
Get trust center capabilities and stack revision
Request Key
Requesting Key Type: [0x04] Trust Center Key
Transport Key
Verify Key
Process key hash to verify the APS key
Confirm Key

Figure 6: **Joining direct to Trust Center.**

A similar process occurs when the device joins through a parent device that is not the TC. The parent device sends APS Update device commands to the TC to notify about the new device and from then the parent device only relays the frames between the joining device and the TC as illustrated in Figure 7.

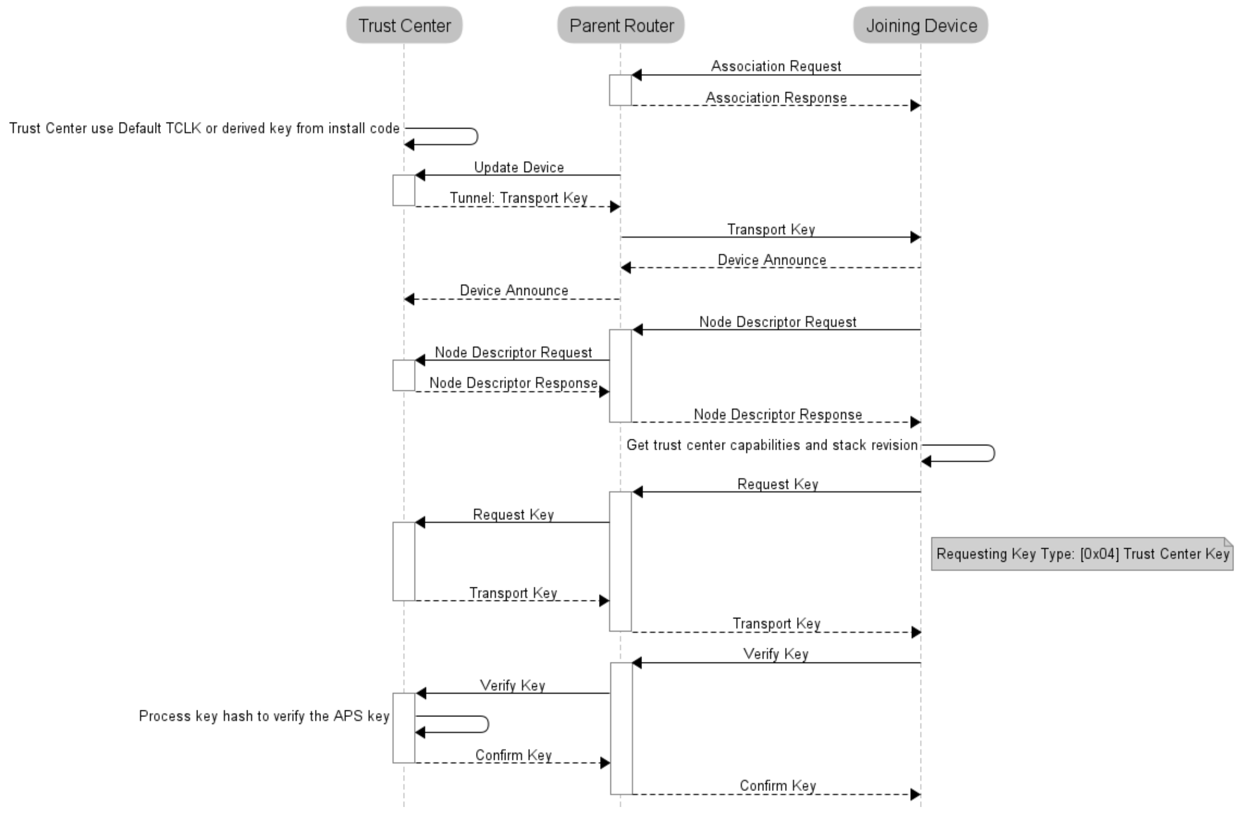


Figure 7: **Joining when parent is not the Trust Center.**

10.6.2 Unsecure join to a distributed network

Once the transport key is received by the joining device, it will proceed to check the source address of this transport key command. In this case the 64-bit IEEE address will be FF's, indicating that this is a distributed network. In this case there are no additional procedures to perform updates of keys since there is no TC that can handles this. The joining process into a Z3.0 Distributed network is illustrated in Figure 8.

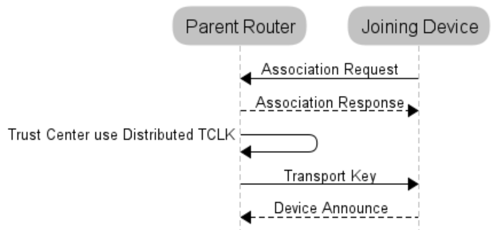


Figure 8: Distributed security joining.

In this case the joining device will attempt up to `BDBC_REC_SAME_NETWORK_RETRY_ATTEMPS` to join this network, if it cannot be authenticated (receive the network key), then it will try the next network in the network descriptor list.

10.6.3 Z-Stack security considerations

10.6.3.1 For TC devices

Trust center devices have a TCLK manager which stores the APS secure information related to a particular joining device (IEEE address, frame counters, key, key status). Each entry is defined by the structure `APSME_TCLKDevEntry_t` defined in APSMEDE.h. These entries are store in Nv and the number of entries that can be stored is defined by `ZDSECMGR_TC_DEVICE_MAX`, defined in *ZDSecMgr.h*. An entry is created for all joining devices at the moment in which the TC sends the network key to the joining deviceThis limits the number of devices in the network to the number of entries that the TC have. A TCLK entry is also used when an Install Code is introduced to the TC for a joining device, and the Install Code key is saved in a separate table of Nv which size is controlled by `ZDSECMGR_TC_DEVICE_IC_MAX` and defined in *ZDSecMgr.h*. The TC frees the Install Code key entry from Nv whenever the related joining device completes the TCLK exchange, leaving this entry free for another device to use the Install Code entry, but it keeps using a TCLK entry. Since the TCLK entries are used to keep track of the APS Key and this is not updated from the Global Default Centralized Key by legacy devices (R20 or before), it does not make sense to keep TCLK entries for those devices, so the TC depending on the configuration of `bdbTrustCenterRequireKeyExchange` (10.3.1.3) will kick the devices out of the network that do not complete the TCLK exchange process and erase the TCLK entry associated to it or will leave it into the network but still will erase the TCLK entry for this device. This optimization allows a Z3.0 TC device to have up to `ZDSECMGR_TC_DEVICE_MAX` Z3.0 devices in the network and as many legacy devices can join the network, being restricted by any other parameter or topology configuration.

10.6.3.2 For joining devices

When a device is factory new and receives the APS Transport Key command, it loads the key to be attempted for a centralized network (Install Code if loaded through BDB API or Global Default Centralized Key if no Install Code is set). If the decryption fails, Z-Stack automatically will attempt with the Global Default Distributed Key. This is because the joining device cannot know which kind of network is being joined, until it processes the content of the Transport Key command.

The secure procedures to join Centralized or Distributed networks are already implemented by the BDB layer.

Joining devices must consider that the APS TCLK exchange will involve the reading/writing to Nv of the APS security material by the TC, so if multiple devices are meant to be commissioned at the same time as Factory New a jitter must be implemented to allow the TC to process the joining procedures of all the devices.

10.7 Touchlink joining

Touchlink commissioning is a distributed security joining that requires physical proximity and uses its own preconfigured link key. For this procedure, a touchlink initiator starts a scan request over all enabled channels looking for a target, if a target responds and is selected it will be asked to form a new network for the initiator or join to the initiator network.

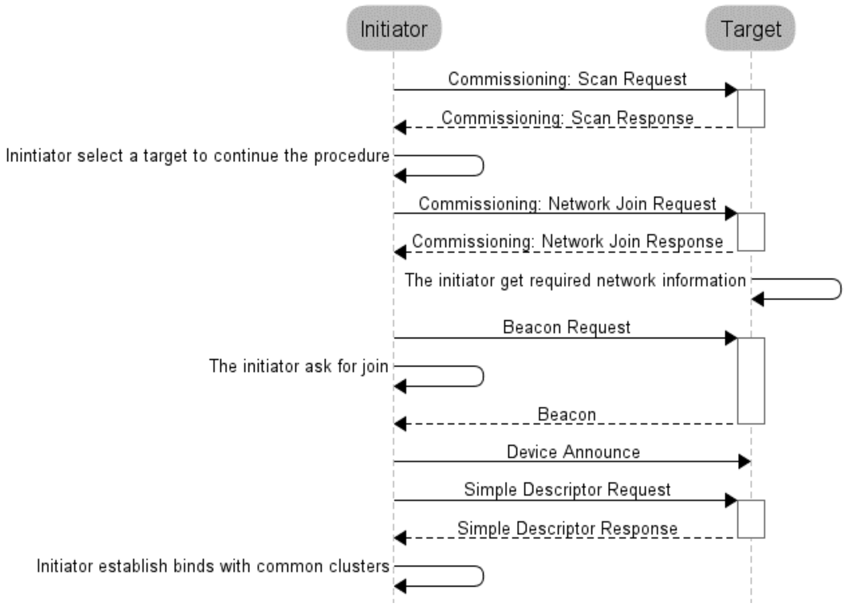

Initiator
Target
Commissioning: Scan Request
Commissioning: Scan Response
Inintiator select a target to continue the procedure
Commissioning: Network Join Request
Commissioning: Network Join Response
The initiator get required network information
Beacon Request
The initiator ask for join
Beacon
Device Announce
Simple Descriptor Request
Simple Descriptor Response
Initiator establish binds with common clusters

Figure 9: Asking to join with Touchlink commissioning.

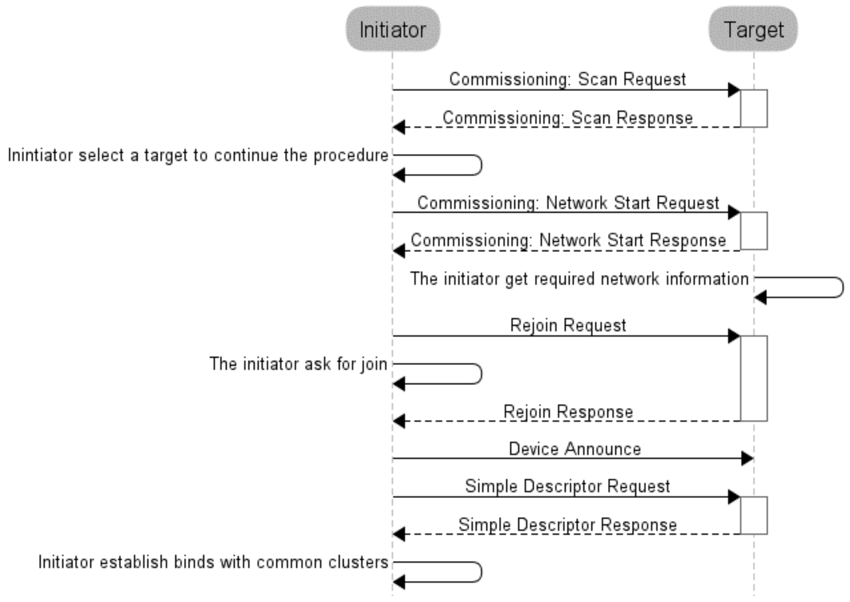

Initiator
Target
Commissioning: Scan Request
Commissioning: Scan Response
Inintiator select a target to continue the procedure
Commissioning: Network Start Request
Commissioning: Network Start Response
The initiator get required network information
Rejoin Request
The initiator ask for join
Rejoin Response
Device Announce
Simple Descriptor Request
Simple Descriptor Response
Initiator establish binds with common clusters

Figure 10: Asking to start network with Touchlink commissioning.

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 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. Application overview

11.1 Introduction

The subsequent sections refer to the Z-Stack™ 3.0 Sample Applications. Each one of the Z-Stack Sample Applications provides a head-start to using the TI distribution of the ZigBee Stack in order to implement a specific Application Object.
The subsequence sections are intended as a generic overview. For a hands-on user's guide, please refer to [3].
Each sample application uses the minimal subset of BDB Public Interfaces that it would take to make a Device reasonably viable in a ZigBee network according to the BDB specification. In addition, all sample applications utilize the essential OSAL API functionality: inter and intra-task communication, by sending and receiving messages, setting and receiving task events, setting and receiving timer callbacks, using dynamic memory, as well as others. In addition, every sample application makes use of the HAL API functionality, e.g. a user interface with buttons and menus on an LCD. Thus, any sample application serves as a fertile example from which to copy-and-paste, or as the base code of the user's application to which to add additional Application Objects.
Any Application Object must sit overtop of a unique Endpoint, and any Endpoint is defined by a Simple Descriptor. The numbers used for the Endpoints in the Simple Descriptors have been chosen arbitrarily.
Each sample application instantiates only one Application Object related to the specific sample application, but it also may contain additional Application Objects in other Endpoints such as GP endpoint (242) for ZC and ZR devices, TL Endpoint (13) for those devices that supports this optional commissioning method. Keep in mind that when instantiating more Application Objects in the same Device, each Application Object must implement a unique Endpoint number, avoiding using ZigBee reserved endpoints (0 or any in the range 242 to 255) and other Endpoints already used by Z-Stack
The following paragraphs describe the OSAL Tasks:

11.1.1 Initialization

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

11.1.2 Organization

As described in the Z-Stack OSAL API [1] document, the OSAL implements a cooperative, round-robin task servicing loop. Each major sub-system of the Z-Stack runs as an OSAL Task. The user must create at least one OSAL Task in which their application will run. This is accomplished by adding their task to the task array [tasksArr defined in OSAL_"Application Name".c] and calling their application's task initialization function in osalInitTask(). The sample applications clearly show how the user adds a task to the OSAL system.

11.1.3 Task Priority

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

11.1.4 System Services

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

11.1.5 Application Design

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

11.1.5.1 One OSAL Task per many Application Objects

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

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

11.1.5.2 One OSAL Task per one Application Object

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

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

11.1.6 Mandatory Methods

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

11.1.6.1 Task Initialization

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

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

11.1.6.2 Task Event Handler

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

11.1.7 Mandatory Events

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

11.1.7.1 SYS_EVENT_MSG (0x8000)

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

11.1.7.1.1 AF_DATA_CONFIRM_CMD

This is an indication of the over-the-air result for each data request that is successfully initiated by invoking `AF_DataRequest()`. `ZSuccess` confirms that the data request was successfully transmitted over-the-air. If the data request was made with the `AF_ACK_REQUEST` flag set, then the `ZSuccess` confirms that the message was successfully received at the final destination. Otherwise, the `ZSuccess` only confirms that the message was successfully transmitted to the next hop.
The next hop device may or may not be the destination device of a data request. Therefore, the AF data confirmation of delivery to the next hop must not be misinterpreted as confirmation that the data request was delivered to the destination device. Refer to the discussion of delivery options (`AF_ACK_REQUEST` in the Z-Stack API).

11.1.7.1.2 AF_INCOMING_MSG_CMD

This is an indication of an incoming AF message.

11.1.7.1.3 KEY_CHANGE

This is an indication of a key press action.

11.1.7.1.4 ZDO_STATE_CHANGE

This is an indication that the network state has changed. The supported states now are: `DEV_HOLD, DEV_INIT, DEV_END_DEVICE, DEV_ROUTER, DEV_ZB_COORD` and `DEV_NWK_ORPHAN`. Other states may not be accurate. It is recommended to base the application behavior on the BDB notifications or to migrate the existing application handling to the notifications that applies. The notifications are described in section 15.1

11.1.7.1.5 ZDO_CB_MSG

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

11.2 Common Application Framework / Program Flow

This section describes the initialization and main task processing concepts that all sample applications share. This section should be read before moving on to the sample applications. For code examples in this section, we will use “SampleLight”. provided in the Z-Stack 3.0 installer.

11.2.1 Initialization

During system power-up and initialization, the task’s initialization function will be invoked (see 11.1.6.1). The structure of this function is explained below, together with some code examples:

```
   zclSampleLight_TaskID = task_id;
```

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

```
   bdb_RegisterSimpleDescriptor ( & zclSampleLight_SimpleDesc );
```

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

```
   RegisterForKeys( zclSampleLight_TaskID );
```

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

11.2.2 Event Processing

Whenever an OSAL event occurs for the sample application, the event processing functions (see 11.1.6.2), will be invoked in turn from the OSAL task processing loop. The parameter to the task event handler is a 16-bit bitmask; one or more bits may be set in any invocation of the function. If more than one event is set, it is strongly recommended that a task should only act on one of the events (probably the most time-critical one, and almost always, the SYS_EVENT_MSG as the highest priority one). The unhandled event will cause a new invocation of this task event handler after all the other handlers get the chance to process their events.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Whenever the network state changes, all tasks are notified with the system event message `ZDO_STATE_CHANGE.` Refer to section 11.1.7.1.4 for supported states.

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

Notice that the design of the OSAL messaging system requires that the receiving task re-cycle the dynamic memory allocated for the message.
The task’s event processing function should try to get the next pending SYS_EVENT_MSG message:

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

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

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

12. The Sample Applications

12.1 Introduction

Z3.0 has removed the concept of profiles and has created a single device definition specification that will mandate the clusters that specific device types must support. At the moment of the release of this documents and Z-Stack, the transition from the devices from the different profiles to Z3.0 is not completed. From the set of sample applications only SampleSwitch and SampleLight have a Z3.0 definition. The other Sample Applications (Doorlock, Doorlock Controller, Temperature Sensor and Thermostat) devices still rely on the ZHA profile definition. Eventually they will be updated to Z3.0 as the clusters for those are converted to Z3.0. These sample applications will still be a good starting point to update to Z3.0, since the changes from the ZHA device definition to Z3.0 device definition will rely on the application.

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

12.2 Initialization

As described in 11, the user must add at least one task (to the OSAL Task System) to service the one or many Application Objects instantiated. But when implementing an Application Object, a task dedicated to the ZCL must also be added (see the “OSAL Tasks” section) in the order shown in the Sample Applications (i.e. before any other tasks using ZCL).
In addition to the user task initialization procedure studied in the generic example in the previous section, the initialization of an Application Object also requires steps to initialize the ZCL. The following Sample Applications (Light and Switch) implement this additional initialization in the task initialization function (e.g. `zclSampleLight_Init()`) as follows.

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

12.3 Software Architecture

Applications send data through various send functions (*e.g. zclGeneral_SendOnOff_CmdToggle*). Applications receive data in callback functions after they register the callbacks. Each cluster has its own register and set of callback functions (*e.g. zclSampleLight_OnOffCB*).
The way Z-Stack communicates with other devices can be described by Figure 11. Clients initiate a command or request a response. Servers act on a command or deliver the response.

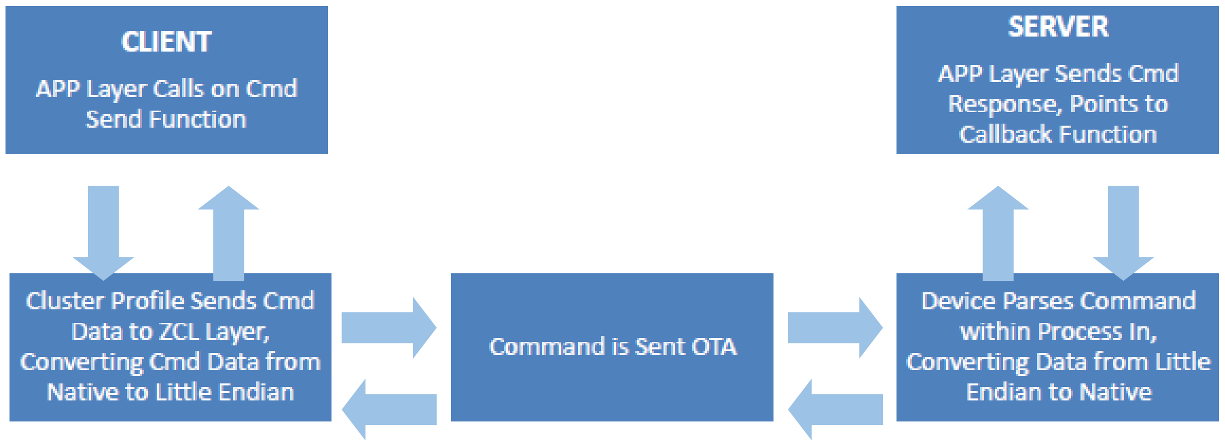

CLIENT
APP Layer Calls on Cmd Send Function
SERVER
APP Layer Sends Cmd Response, Points to Callback Function
Cluster Profile Sends Cmd Data to ZCL Layer, Converting Cmd Data from Native to Little Endian
Command is Sent OTA
Device Parses Command within Process In, Converting Data from Little Endian to Native

Figure 11: Command Flowchart

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

As an example of a callback function, if a client sends a BasicReset command to a server, the application's registered BasicReset callback function on the server side is called (*e.g. zclSampleDoorLockController_BasicResetCB)*, which will cause the device to reset all the attributes in all clusters supported by the application to their default values.

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

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

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

12.4 File Structure

The following directories hold most of the Z3.0 sample applications related code:

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

1. IAR workspace files for Light project:

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

2. Source files for Light project:

   \Projects\zstack\HomeAutomation\SampleLight\Source

3. IAR workspace files for Switch project:

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

4. Source files for Switch project:

   \Projects\zstack\HomeAutomation\SampleSwitch\Source

12.4.2 DoorLock Sample Application (DoorLock and DoorLockController).

1. IAR workspace files for DoorLock project:

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

2. Source files for DoorLock project:

   \Projects\zstack\HomeAutomation\SampleDoorLock\Source

3. IAR workspace files for DoorLockController project:

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

4. Source files for DoorLockController project:

   \Projects\zstack\HomeAutomation\SampleDoorLockController\Source

12.4.3 Temperature Sensor Sample Application (Thermostat, TemperatureSensor).

1. IAR workspace files for Thermostat project:

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

2. Source files for Thermostat project:

   \Projects\zstack\HomeAutomation\SampleThermostat\Source

3. IAR workspace files for TemperatureSensor project:

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

4. Source files for TemperatureSensor project:

   \Projects\zstack\HomeAutomation\SampleTemperatureSensor\Source

12.4.4 Additional Project Files.

1. Source files common to all Z3.0 projects:

   \Projects\zstack\HomeAutomation\Source

2. ZigBee Cluster Library source code:

   \Components\stack\zcl

12.5 Sample Light Application

12.5.1 Introduction

This sample application can be used to turn on/off the LED 1 on the device using the On/Off cluster commands.

12.5.2 Modules

The Sample Light application consists of the following modules (on top of the ZigBee stack modules):
OSAL_SampleLight.c - functions and tables for task initialization.
zcl_samplelight.c – main application function that has init and event loop function.
zcl_samplelight.h – header file for application module.
zcl_samplelight_data.c – container for declaration of attributes, clusters, simple descriptor.

12.6 Sample Switch Application

12.6.1 Introduction

This sample application can be used as the Light Switch to turn on/off the LED 1 on a device running the Sample Light application.

12.6.2 Modules

The Sample Switch application consists of the following modules (on top of the ZigBee stack modules):
OSAL_SampleSw.c - functions and tables for task initialization.
zcl_samplesw.c – main application function that has init and event loop function.
zcl_samplesw.h – header file for application module.
zcl_samplesw_data.c – container for declaration of attributes, clusters, simple descriptor.

12.6.3 Sample Switch OTA Demonstration Build Configurations

12.6.3.1 Introduction

The Sample Switch Application work space contains several build configurations that demonstrate firmware upgrade using the ZigBee OTA (i.e. over the air) feature. When using OTA, to upgrade the firmware, up to two FW images may reside in the flash memory at a single time: The active image, and the downloaded image, that will be activated upon next power on.
For details of OTA update functionality see [4].

12.6.3.2 Modules

This build configuration adds the following modules the Sample Switch application:
hal_ota.c - OTA utility functions that are platform specific.
hal_ota.c – header file for OTA utility functions.
zcl_ota.c – event handler for base OTA ZCL event handling.
zcl_ota.h – header file for OTA even handler module.
ota_common.h – Contains definitions required when utilizing OTA.

12.7 Sample Door Lock Application

12.7.1 Introduction

This sample application can be used as a Door Lock, and can receive Door Lock cluster commands, like toggle lock/unlock, and set master PIN, from a Door Lock Controller device. It also has preconfigured attribute reporting of the door lock state.

12.7.2 Modules

The Sample Door Lock application consists of the following modules (on top of the ZigBee stack modules):
OSAL_SampleDoorLock.c - functions and tables for task initialization.
zcl sampledoorlock.c – main application function that has init and event loop function.
zcl sampledoorlock.h – header file for application module.
zcl sampledoorlock_data.c – container for declaration of attributes, clusters, simple descriptor.

12.8 Sample Door Lock Controller Application

12.8.1 Introduction

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

12.8.2 Modules
The Sample Door Lock Controller application consists of the following modules (on top of the ZigBee stack modules):
OSAL_SampleDoorLockController.c - functions and tables for task initialization.
zcl_sampledoorlockcontroller.c – main application function that has init and event loop function.
zcl_sampledoorlockcontroller.h – header file for application module.
zcl_sampledoorlockcontroller_data.c – container for declaration of attributes, clusters, simple descriptor.

12.9 Sample Thermostat Application

12.9.1 Introduction
This sample application can be used to receive measurements of temperature from the Temperature Sensor device

12.9.2 Modules
The Sample Thermostat application consists of the following modules (on top of the ZigBee stack modules):
OSAL_SampleThermostat.c - functions and tables for task initialization.
zcl_samplethermostat.c – main application function that has init and event loop function.
zcl_samplethermostat.h – header file for application module.
zcl_samplethermostat_data.c – container for declaration of attributes, clusters, simple descriptor.

12.10 Sample Temperature Sensor Application

12.10.1 Introduction
This sample application can be used as a Temperature Sensor device to send measurements of temperature to the Thermostat device. Locally the temperature can be increased/decreased and also manually send the current temperature to the Temperature Sensor. It also has preconfigured attribute reporting of the temperature meassurement state.

12.10.2 Modules
The Sample Temperature Sensor application consists of the following modules (on top of the ZigBee stack modules):
OSAL_SampleTemperatureSensor.c - functions and tables for task initialization.
zcl_sampletemperaturesensor.c – main application function that has init and event loop function.
zcl_sampletemperaturesensor.h – header file for application module.
zcl_sampletemperaturesensor_data.c – container for declaration of attributes, clusters, simple descriptor.

12.11 Main Functions
All sample applications have the same architecture and have the same basic modules:
`<Sample_App>_Init()` – Initializes the Application task:
- Registers the application’s endpoint and its simple descriptor.
- Registers the ZCL General Cluster’s callbacks.
- Registers the application’s attribute list.
- Register the Application to receive the unprocessed Foundation command/response messages.
- Register the commissioning status callback.
- Register for all key-press events.
- Registers a test end-point
- Registers for the following ZDO message:
    - Match_Desc_rsp

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

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

`<Sample_App>_ProcessZDOMsgs()` – Process the following response messages:

- Match_Desc_rsp
- All other ZDO messages the Sample Application registered for.

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

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

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

`bdbRegisterCommissioningStatusCB()` – This callback will be informed of BDB commissioning process events and status.

13. Using the sample applications as base for new applications

The Z3.0 Sample Applications are intended to be used as foundations for the user's applications. Modifying them will usually consist of the following steps:

1. Copy the Sample Application of your choice to a new folder, and name it according to your new application:
   Copy
   Project\zstack\HomeAutomation\SampleLight\*.*
   To
   Project\zstack\HomeAutomation\YourApp\*.*
2. Rename the files to match your application's name:
   Under Projects\zstack\HomeAutomation\YourApp\Source, rename the files as follows:
   - OSAL_SampleLight.c → OSAL_YourApp.c
   - zcl_samplelight.c → zcl_yourapp.c

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

   Delete all the other files and subfolders from this folder, if there are any.
4. Using a text editor, replace all occurrences of "SampleLight" (case insensitive) to "YourApp", in YourApp.ewp and YourApp.eww.

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

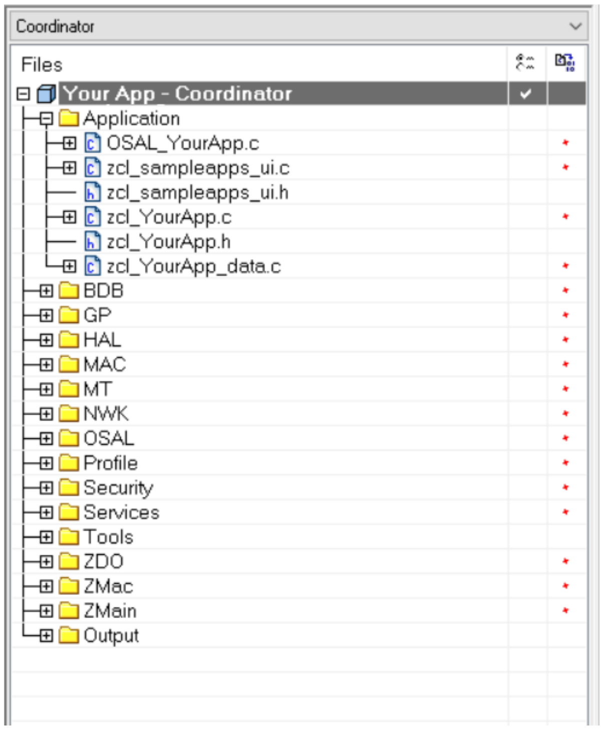


Figure 12: Application files within YourApp

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

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

14. Clusters, Commands and Attributes

Each application supports a certain number of clusters. Think of a cluster as an object containing both methods (commands) and data (attributes).
Each cluster may have zero or more commands. Commands are further divided into Server and Client-side commands. Commands cause action, or generate a response.
Each cluster may have zero or more attributes. All of the attributes can be found in the *zcl_sampleapp_data.c* file, where "sampleapp" is replaced with the given sample application (e.g. *samplesw_data.c* for the sample on/off light switch). Attributes describe the current state of the device, or provide information about the device, such as whether a light is currently on or off.
All clusters and attributes are defined either in the ZigBee Cluster Library specification.

14.1 Attributes

Attributes are found in a single list called `zclSampleApp_Attrs[ ]`, in the zcl_sampleapp_data.c file. Each attribute entry is initialized to a type and value, and contains a pointer to the attribute data. Attribute data types can be found in the ZigBee Cluster Library.
The attributes must be registered using the `zcl_registerAttrList ( )` function during application initialization, one per application endpoint.
Each attribute has a data type, as defined by ZigBee (such as `UINT8`, `INT32`, etc…). Each attribute record contains an attribute type and a pointer to the actual data for the attribute. Read-only data can be shared across endpoints. Data that is unique to an endpoint (such as the OnOff attribute state of the light) should have a unique C variable.
All attributes can be read. Some attributes can be written. Some attributes are reportable (can be automatically sent to a destination based on time or change in attribute via the attribute reporting functionality). Some attributes are saved as part of a "scene" that can later be recalled to set the device to a particular state (such as a light on or off). The attribute access is controlled through a field in the attribute structure.
To store an attribute in non-volatile memory (to be preserved across reboots) refer to section 9.6.3.

14.2 Adding an Attribute Example

To add an additional attribute to a project, refer to the attribute's information within the ZCL Specification [5]. Using the DoorLock cluster as an example, the following will show how to add the "Max PIN Code Length" attribute to the DoorLock project. This process can be replicated across all Z3.0 sample projects.
All attributes in use by an application are defined within the project source file's zcl_sampleapplication_data.c file. For this DoorLock example, this data file is: *zcl_sampledoorlock_data.c*. Locate the section defined as *Attribute Definitions* and include the "Max PIN Code Length" attribute using the format:

```
{
  ZCL_CLUSTER_ID_CLOSURES_DOOR_LOCK,
  { // Attribute record
    ATTRID_DOORLOCK_NUM_OF_MAX_PIN_LENGTH,
    ZCL_DATATYPE_UINT8,
    ACCESS_CONTROL_READ,
    (void *)&zclSampleDoorLock_NumOfMaxPINLength
  }
},
```

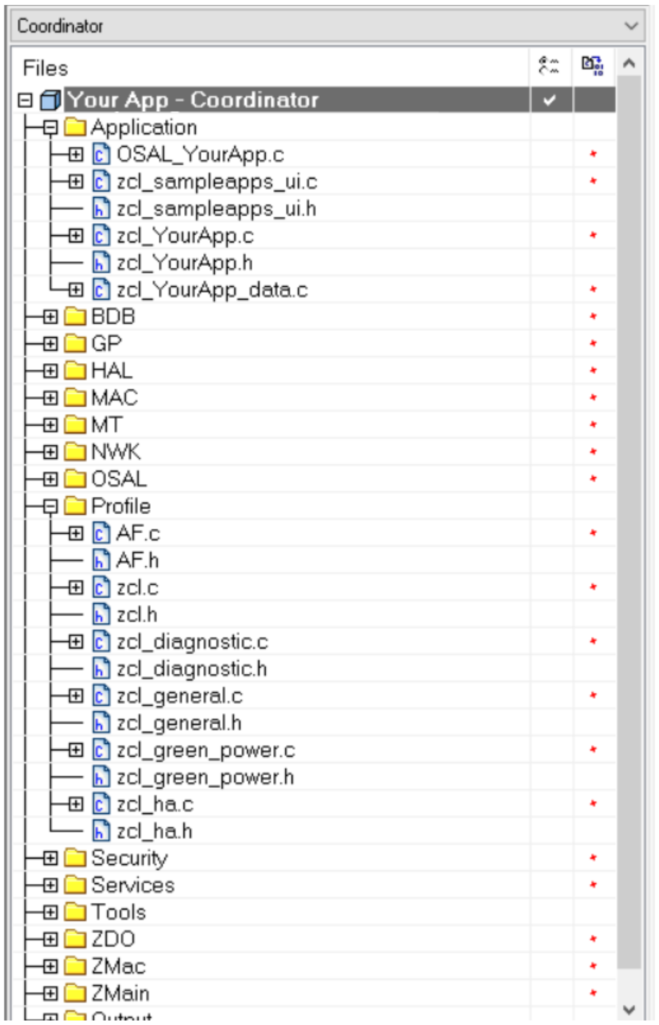
Line 2 represents the cluster ID, line 4 represents the attribute ID, line 5 the data type, line 6 the read/write attribute, and line 7 the pointer to the variable used within the application.
The cluster ID can be retrieved from the *zcl.h* file, the attribute ID can be found within the (in this case) *zcl_closures.h* file, and the remaining information from the ZCL Specification [5].
By including the attribute within this list, devices are able to interact with the attributes on other devices. This addition in the attribute list must be reflected in the `SAMPLEDOORLOCK_MAX_ATTRIBUTES` macro in the *zcl_sampledoorlock.h* file. Also within that file, define the external variable using proper coding conventions:

```
       extern uint8 zclSampleDoorLock_NumOfMaxPinLength
```

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

14.3 Initializing Clusters

For the application to interact with a cluster the cluster's compile flag must be enabled (if applicable to the cluster) in the project's configuration and the cluster's source file must be added to the project's Profile folder within the IAR Workspace. An example of this can be seen in Figure 13. See *f8wZCL.cfg* for a list of cluster compile flags. Once enabled, the cluster's callbacks can be registered within the application (refer to section 14.5).



```
Coordinator
Files
⊟ Your App - Coordinator
├⊟ Application
│ ├⊞ OSAL_YourApp.c
│ ├⊞ zcl_sampleapps_ui.c
│ ├── zcl_sampleapps_ui.h
│ ├⊞ zcl_YourApp.c
│ ├── zcl_YourApp.h
│ └⊞ zcl_YourApp_data.c
├⊞ BDB
├⊞ GP
├⊞ HAL
├⊞ MAC
├⊞ MT
├⊞ NWK
├⊞ OSAL
├⊟ Profile
│ ├⊞ AF.c
│ ├── AF.h
│ ├⊞ zcl.c
│ ├── zcl.h
│ ├⊞ zcl_diagnostic.c
│ ├── zcl_diagnostic.h
│ ├⊞ zcl_general.c
│ ├── zcl_general.h
│ ├⊞ zcl_green_power.c
│ ├── zcl_green_power.h
│ ├⊞ zcl_ha.c
│ └── zcl_ha.h
├⊞ Security
├⊞ Services
├⊞ Tools
├⊞ ZDO
├⊞ ZMac
├⊞ ZMain
```

Figure 13: List of Cluster Source Files in Profile Folder

14.4 Cluster Architecture

All clusters follow the same architecture.
The clusters take care of converting the structures passed from native format to over-the-air format, as required by ZigBee. All application interaction with clusters takes place in native format.
They all have the following functions:

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

There is usually one send function for each command. The ***send*** function has either a set of parameters or a specific structure for the command.
If the application has registered callback functions, then the ***ProcessIn*** will direct the command (after it's converted to native form) to the application callback for that command.

14.5 Cluster Callbacks Example

Callbacks are used so that the application can perform the expected behavior on a given incoming cluster command. It is up to the application to send a response as appropriate. Z-Stack provides the parsing, but it is up to the application to perform the work.
A cluster's callback functions are registered within the application's initialization function by including the application's endpoint and a pointer to the callback record within a register commands callback function. Figure 14 shows an example of the general cluster's callback record list. The commands are registered to their respective callback functions as defined within the cluster's profile.
As an example, once a BasicReset command reaches the application layer on a device, the cluster's callback record list points the command to the BasicReset callback function: `zclSampleLight_BasicResetCB`. The application reset command can then reset all data back to Factory New defaults.
The callback function in an application provides additional processing of a command that is specific to that application. These callback functions work alongside the response to the incoming command, if a response is appropriate.

```
static zclGeneral_AppCallbacks_t zclSampleLight_CmdCallbacks =
{
  zclSampleLight_BasicResetCB,            // Basic Cluster Reset command
  NULL,                                   // Identify Trigger Effect command
  zclSampleLight_OnOffCB,                 // On/Off cluster commands
  NULL,                                   // On/Off cluster enhanced command Off with Effect
  NULL,                                   // On/Off cluster enhanced command On with Recall Global Scene
  NULL,                                   // On/Off cluster enhanced command On with Timed Off
#ifdef ZCL_LEVEL_CTRL
  zclSampleLight_LevelControlMoveToLevelCB, // Level Control Move to Level command
  zclSampleLight_LevelControlMoveCB,        // Level Control Move command
  zclSampleLight_LevelControlStepCB,        // Level Control Step command
  zclSampleLight_LevelControlStopCB,        // Level Control Stop command
#endif
#ifdef ZCL_GROUPS
  NULL,                                   // Group Response commands
#endif
#ifdef ZCL_SCENES
  NULL,                                  // Scene Store Request command
  NULL,                                  // Scene Recall Request command
  NULL,                                  // Scene Response command
#endif
#ifdef ZCL_ALARMS
  NULL,                                  // Alarm (Response) commands
#endif
#ifdef SE_UK_EXT
  NULL,                                  // Get Event Log command
  NULL,                                  // Publish Event Log command
#endif
  NULL,                                  // RSSI Location command
  NULL                                   // RSSI Location Response command
};
```

Figure 14: Cluster Callbacks Example

14.6 Attribute Reporting Functionality

The Attribute Reporting module takes care of periodically sending the ZCL Report Attributes command messages for all reportable attributes defined in the application. The module also processes the ZCL Configure Reporting and Read Reporting Configuration commands. Multiple independent compilation flags control the reporting functionality, so unneeded functionality can be omitted from the code to save resources.

- To enable BDB ***report sending*** functionality on a device, include the **BDB_REPORTING** compile option.

- To enable BDB **report receiving/processing** functionality, include the **ZCL_REPORT_DESTINATION_DEVICE** compile option.
- To enable configuring reporting parameters of remote devices, include the **ZCL_REPORT_CONFIGURING_DEVICE** compile option.

The ***report sending*** functionality implementation is in *bdb_reporting.c*

The Attribute Reporting functionality was implemented as described in the ZCL document [5], however in order to optimize the number of Report Attributes command messages sent over the air, a consolidation was made for attributes in the same cluster, for all clusters in every endpoint. In other words, this means that all reportable attributes in the same cluster will have only one consolidated Minimum Reporting Interval and Maximum Reporting Interval value. The consolidation approach used to merge the Attribute Reporting Configuration record's Maximum Reporting Interval is to grab the minimum value of all the attribute values of a same cluster, the consolidated Minimum Reporting Interval of the cluster is also the minimum value. Refer to ZCL document [5] section 2.5.11.2.5 for further details on consolidation of reportable attributes.

The Attribute Reporting module automatically looks into the attribute definitions registered in the application for all the attributes with the ACCESS_REPORTABLE flag. Each of these reportable attributes will have a corresponding Attribute Reporting Configuration record later set to some default values. The Attribute Reporting module automatically starts or stops the reporting of the attributes in a cluster of an endpoint which bind is added or removed.

In the BDB API (located in the *bdb_interface.h* file) there is a method called `bdb_RepAddAttrCfgRecordDefaultToList` that is used to add each Attribute Reporting Configuration record default values. This method must be called before the device starts the BDB Commissioning. If the applications does not add default values for a given Attribute Reporting Configuration record, then global defaults values will be assign, these global default MACROS are located in *bdb_Reporting.h*.

When the BDB state machine starts commissioning, the Attribute Reporting module either loads the previously saved Attribute Reporting Configuration records from NV or searches the applications attribute definitions to deduce the reportable attributes and construct the necessary Attribute Reporting Configuration records using the defaults values previously added by the application. Then the module will consolidate the reportable attributes in each cluster of every endpoint, in order to trigger the periodic sending of the Report Attributes command messages using the Maximum Reporting Interval values.

At runtime, the Attribute Reporting module listens for Configure Reporting Command messages and will reconsolidate the cluster's Maximum Reporting Interval and Minimum Reporting Interval values given the new Attribute Reporting Configuration records contain in the message. Calls to the `bdb_RepAddAttrCfgRecordDefaultToList` method after the BDB Commissioning has started will have effect on the current Attribute Reporting Configuration records.

In order for the Attribute Reporting module to manage the sending of Report Attributes commands when the attributes changes value, the application must inform the module when any reportable attribute has a new value. This notification must be made by calling the `bdb_RepChangedAttrValue` method of the BDB API. The Attribute Reporting module will get the current value of the attribute from the callback defined in the application attribute definitions, meaning that the new value must be set before calling the notification method.

15. Commissioning

The BDB commissioning method provides a mechanism to invoke a series of procedures that provides the ability to easily connect devices together. Depending on the commissioning methods invoked, devices will perform actions like forming networks, joining existing networks, and binding application endpoints.
The source files that control the commissioning procedures are located in the group of files BDB in IAR projects. The API interface is located in *bdb_interface.h* and described in [1].
The BDB interface provides an API to trigger one or more commissioning procedures defined as follows:

```
bdb_StartCommissioning( mode )
```

where `mode` is the bitmask for the commissioning modes to be executed and defined as:

```
BDB_COMMISSIONING_MODE_INITIATOR_TL         0b00000001
BDB_COMMISSIONING_MODE_NWK_STEERING         0b00000010
BDB_COMMISSIONING_MODE_NWK_FORMATION        0b00000100
BDB_COMMISSIONING_MODE_FINDING_BINDING      0b00001000
```

This commissioning mask is appended to the current commissioning modes being executed. The tasks are also executed with the priority listed before (TL as initiator first, then Nwk steering, then Formation and lastly Finding and Binding). The priority of the tasks are check whenever another tasks is finished, so if TL is requested before Nwk Steering and Formation are executed, then TL initiator will be process after Nwk Steering but before Formation. The tasks can be appended at any time (E. g. in response to a commissioning notification).
There are other commissioning states that the BDB machine state handles as modes, these are `INITIALIZATION` and `PARENT_LOST`. These states should not be directly used by the application.

15.1 BDB notifications

The application must register a callback using `bdb_RegisterCommissioningStatusCB` in which it will receive notifications on the commissioning process performed by the BDB module. The application can trigger another commissioning method upon receiving a certain notification, e.g. a router device may start network steering to search a suitable network and count the number of times this process fails; if this process fails 'x' times in a row, it may decide to change the channel mask to search networks in other channels not attempted or to call for formation and create its own network. The full API is described in [1].
The notifications are called when certain tasks start or when they finish with the resulting status. Some notifications are exclusive to certain logical devices types or do have a different meaning for different logical devices.
Every notification will have a pointer to a structure of type `bdbCommissioningModeMsg_t` which contains the commissioning mode being reported, the status and the mask of the remaining commissioning modes to be executed. The notification definitions can be found in *bdb_interface.h* and the complete table of the notifications and its description can be found in the following table. In the first column are the commissioning modes, note that these macro definitions can be found in the code as `BDB_COMMISSIONING_`*`mode`* where the word "*mode*" must be replaced by the any of the modes found in the first column. The same applies to the second column, which are the statuses of the commissioning mode being reported. The macro definitions in the code can be found as BDB_COMMISSIONING_*status* where the word "*status*" must be replaced by any of the statuses found in the second column.

Commissioning mode (BDB_COMMISSIONING_*mode*)	Status reported (BDB_COMMISSIONING_*status*)	Description
INITIALIZATION	NETWORK_RESTORED	Only send if the device did restore its network parameters. On end devices, if no parent is found with the restored network parameters, a *Parent Lost* mode is with status *No Network* is notified.
NWK_STEERING (for Router and End Devices)	IN_PROGRESS	Notifies when network steering is started (only if the device is not in a network, otherwise reports *success*)
	NO_NETWORK	No suitable network was found in primary channel or secondary channel masks or the joining process did fail in the attempted networks.
	TCLK_EX_FAILURE	The device successfully joined the network, but could not perform the Trust Center Link Key exchange process. The device will reset to factory new after this notification is reported to the application.
	SUCCESS	The device is now on a network and broadcasted a Management Permit Joining ZDO frame.
NWK_STEERING (for Coordinators)	NO_NETWORK	The device is not on a network, so it cannot perform this action.
	SUCCESS	The device is in a network and has broadcasted a Management Permit Joining ZDO frame.
FORMATION	IN_PROGRESS	Notifies when formation process is started.
	SUCCESS	The network has been created successfully.
	FORMATION_FAILURE	The device could not create the network with the given parameters.
FINDING_BINDING	FB_TARGET_IN_PROGRESS	Indicates the start of the Finding and Binding as target. No notification is given by this callback when the process ends.
	FB_INITITATOR_IN_PROGRESS	Indicates the start of the Finding and Binding as Initiator.
	FB_NO_IDENTIFY_QUERY_RESPONSE	After complete the Finding and Binding process as initiator (single attempt of periodic attempt), no identify query responses were received.
	FB_BINDING_TABLE_FULL	During the Finding and Binding process the binding table got full, so the process stops and no additional binds can be added.
	FAILURE	No endpoint was found to perform Finding and Binding, or the endpoint did not have implemented the Identify cluster properly.
TOUCHLINK	TL_TARGET_FAILURE	A node has not joined a network when requested during touchlink.
	TL_NOT_AA_CAPABLE	The initiator is not address assignment capable during touchlink
	TL_NO_SCAN_RESPONSE	No response to a Scan Reques inter-PAN command has been received during touchlink
	TL_NOT_PERMITTED	A touchlink steal attempt was made when a node is already connected to a centralized security network.
PARENT_LOST (Only for End Devices)	NO_NETWORK	This is notified if the end device does lose contact with the parent device or if after initialization it cannot find a parent device in the commissioned network.
	NETWORK_RESTORED	Notification that a suitable parent device got found and the rejoin process was successful.

Table 1: Commissioning status reported by the different commissioning modes

15.2 Initialization procedure

The BDB interface will perform an initialization once per power cycle and is controlled by the global RAM variable `bdb_initialization` and triggered by any call to `bdb_StartCommissioning()` with any commissioning mode mask. The initialization procedure retrieves the network parameters from Nv if the attribute `bdbNodeIsOnANetwork` is `TRUE`. This attribute is also stored in Nv and retrieved from Nv during the initialization process. For coordinator and router devices, a silent rejoin will be performed (the device will resume operations in the network as if it never left, except that it will process parent annce, see section 9.12.3 to see when parent annce is triggered). End devices will restore the network parameters and will try to perform a rejoin on any parent available in the same network only one time. This procedure is illustrated in Figure 15.

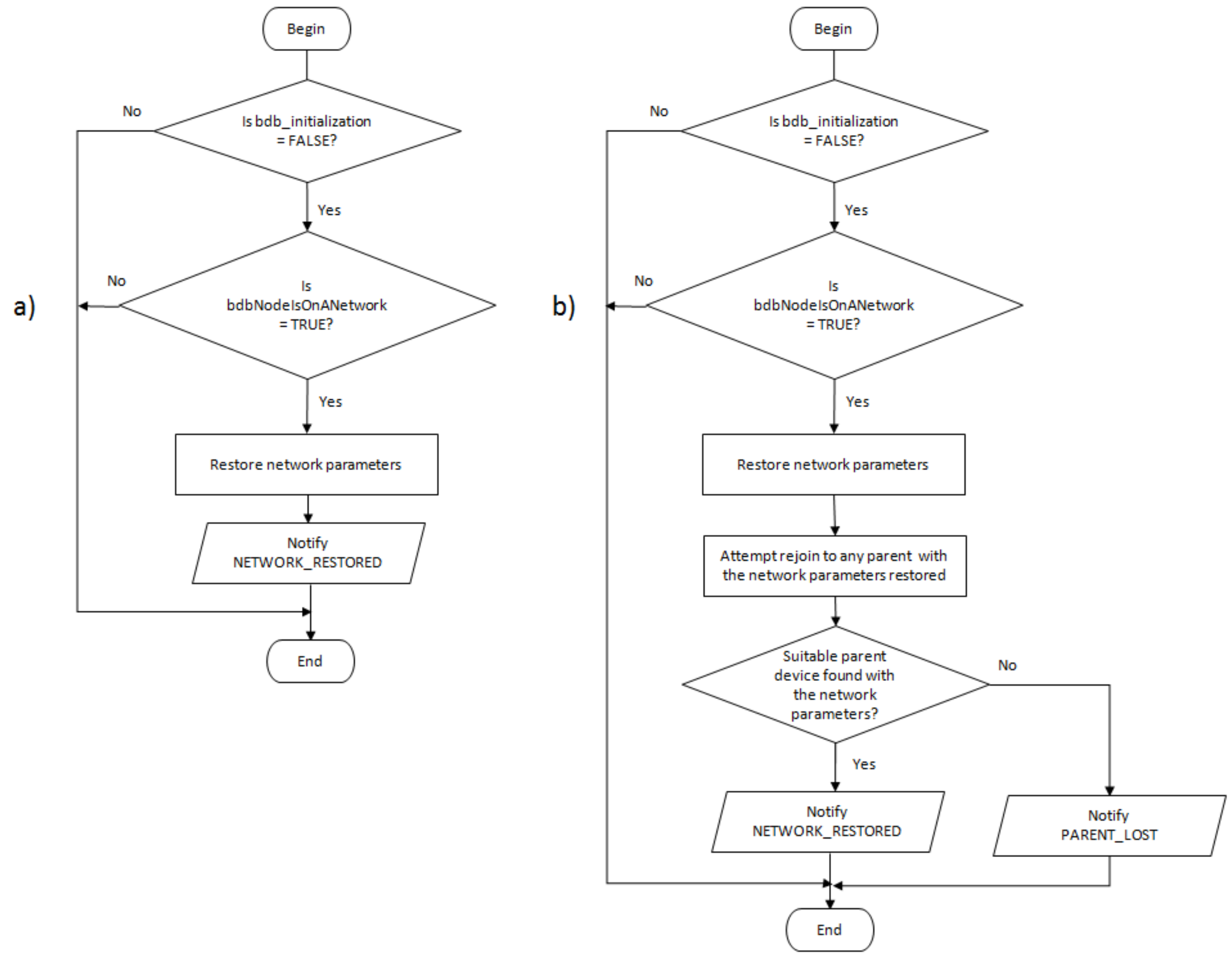

a)
Begin
Is bdb_initialization = FALSE?
No
Yes
Is bdbNodeIsOnANetwork = TRUE?
No
Yes
Restore network parameters
Notify NETWORK_RESTORED
End
b)
Begin
Is bdb_initialization = FALSE?
No
Yes
Is bdbNodeIsOnANetwork = TRUE?
No
Yes
Restore network parameters
Attempt rejoin to any parent with the network parameters restored
Suitable parent device found with the network parameters?
No
Yes
Notify NETWORK_RESTORED
Notify PARENT_LOST
End

Figure 15: Initialization procedures: a) Router and Coordinators, b) End Devices.

Note that if the initialization process fails for an end device it will notify to the application a PARENT_LOST status. Refer to section 15.3 on how to restore the network or section 15.9 to reset the device to factory new.

15.3 Parent Lost

If an end device loses contact with its parent device or is reset while being on a network, the BDB module will notify the application a status of `BDB_COMMISSIONING_PARENT_LOST` after which the end device cannot perform any other commissioning method. The device must either restore its network by finding another parent device or reset to factory new and then be commissioned again. To restore the network the device must call `bdb_ZedAttemptRecoverNwk`, this will cause the device to perform a single active scan in the same channel in which it was part of the network to search for any suitable parent (same Extended PANID and child device capacity). This means that the device will only send a single beacon request and if no suitable parent device is found, another notification BDB_COMMISSIONING_PARENT_LOST is sent to the application. The application is responsible for attempting to restore the network, but it is recommended to have a period in which the attempts have a short interval, then goes to a larger interval, to reduce the power consumption. If Finding and Binding was in progress while the device lost its parent, it will keep running and will resume its operation for the time left after the device restores its operation

15.4 Network steering procedure for a node on a network

If network steering is invoked by a device that is already on a network (bdbNodeIsOnANetwork set to TRUE), it will broadcast a permit joining request for 180 seconds (BDBC_MIN_COMMISSIONING_TIME), after which the device will notify BDB_COMMISSIONING_SUCCESS.

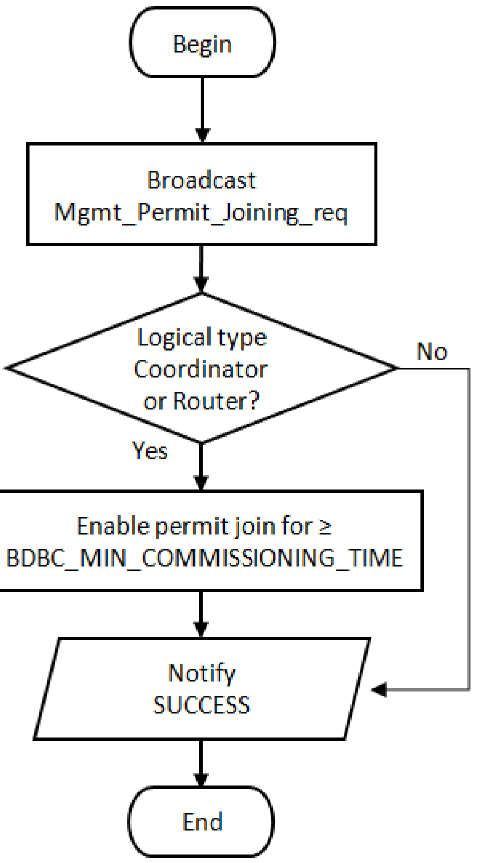

Begin
Broadcast Mgmt_Permit_Joining_req
Logical type Coordinator or Router?
No
Yes
Enable permit join for ≥ BDBC_MIN_COMMISSIONING_TIME
Notify SUCCESS
End

Figure 16: Network steering procedure for a node on a network.

15.5 Network steering procedure for a node not on a network

This procedure is performed when Network Steering is requested and the device is not on a network (bdbNodeIsOnANetwork set to FALSE). This will cause the device to start looking for suitable networks to join. The procedure is illustrated in Figure 17 and described as follows:

1. The device will perform a scan in all channels defined in BDB_DEFAULT_PRIMARY_CHANNEL_SET, searching for any suitable network and creating a network descriptor list of the networks found.
    a. The application may filter the networks found by registering a callback function with bdb_RegisterForFilterNwkDescCB(). The application callback will receive the

network descriptor list containing all networks found, then the application can use `bdb_nwkDescFree()` to release network descriptors of networks that it will not attempt to join (E.g only known networks by Extended PAN ID want to be attempted).

   b. If no suitable networks are found or the device cannot perform joining on the networks found (association was not successful or could not get the network key), the device will proceed to perform the same steps but with the channel mask defined in `BDB_DEFAULT_SECONDARY_CHANNEL_SET`.
   c. Only non-zero channel masks are used for network discovery.

2. The BDB state machine will try to perform association and authentication in the suitable networks discovered using the security keys for Centralized networks (default key or Install Code) or Distributed networks as defined in section 10. For Centralized networks it will also perform the TCLK exchange.
3. If the joining procedure is completed the joining device will broadcast a permit joining request to refresh the joining timeout for other devices trying to join simultaneously. Is up to the network manager to close the network for joining if desired, by sending a permit join request with timeout = 0..

Each step will report to the application a status in the callback function register with `bdb_RegisterCommissioningStatusCB()`. The possible statuses for this commissioning process is described in Table 1.

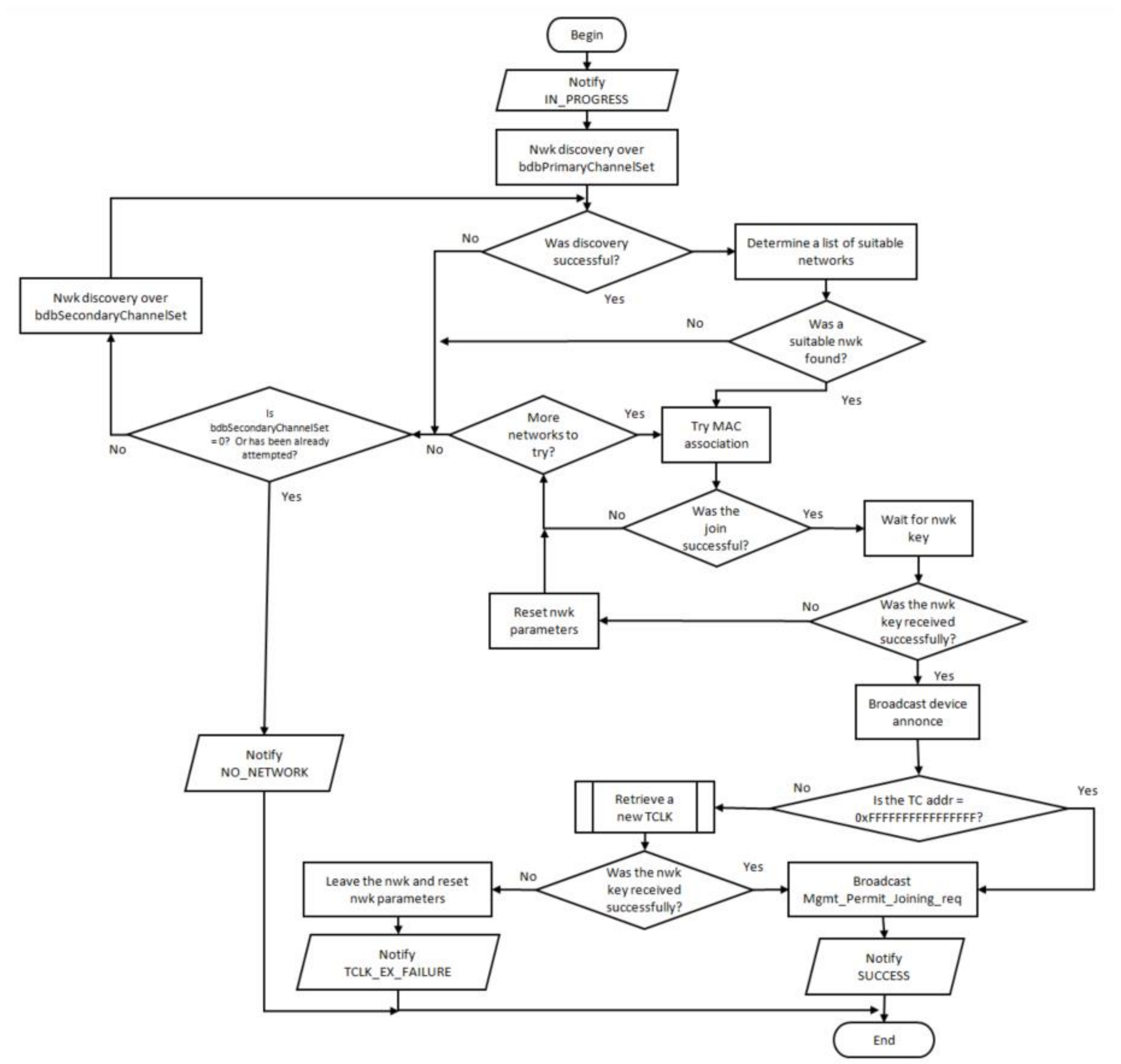

Begin
Notify IN_PROGRESS
Nwk discovery over bdbPrimaryChannelSet
Was discovery successful?
No
Yes
Determine a list of suitable networks
Was a suitable nwk found?
No
Yes
Nwk discovery over bdbSecondaryChannelSet
Is bdbSecondaryChannelSet = 0? Or has been already attempted?
No
Yes
More networks to try?
No
Yes
Try MAC association
Was the join successful?
No
Yes
Wait for nwk key
Reset nwk parameters
Was the nwk key received successfully?
No
Yes
Broadcast device annonce
Notify NO_NETWORK
Is the TC addr = 0xFFFFFFFFFFFFFFFF?
No
Yes
Retrieve a new TCLK
Was the nwk key received successfully?
No
Yes
Leave the nwk and reset nwk parameters
Broadcast Mgmt_Permit_Joining_req
Notify TCLK_EX_FAILURE
Notify SUCCESS
End

Figure 17: Network steering procedure for a node not on a network.

15.6 Network formation

This procedure defines the steps to take when a device with formation capabilities is instructed to form a network (coordinator or router). If an end device is instructed to perform formation, then it will report a failure.

The formation process for devices with formation capabilities consists of a first attempt to create the network in any of the channels selected in the primary channel mask, and if for any reason it cannot perform the formation in those channels (channel mask invalid or selected PAN ID already found in the same channel) the device will try to perform formation in the secondary channel mask. If both of these procedures fail, it will report a `BDB_COMMISSIONING_FORMATION_FAILURE` to the application. If formation is performed successfully then a `BDB_COMMISSIONING_FORMATION_SUCCESS` is sent instead. Note that this procedure does not open the network for joining, but the application may trigger the steering procedure after this to open it.

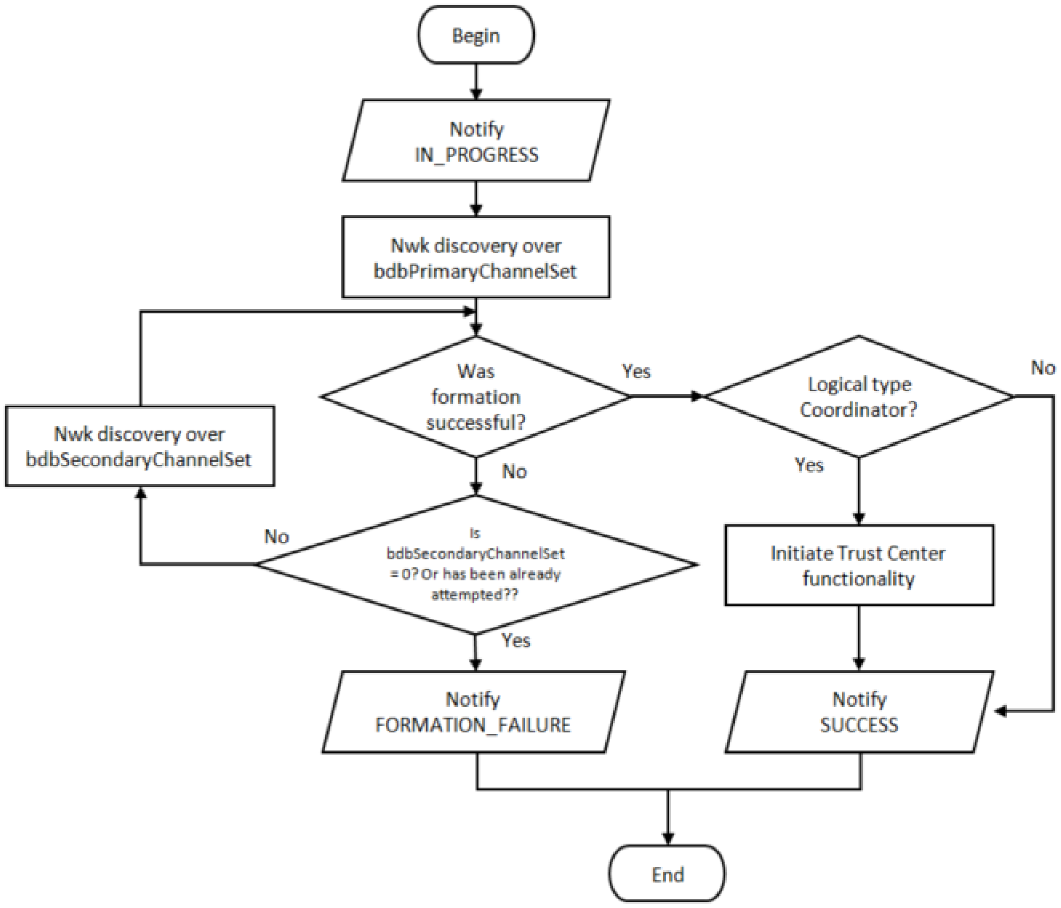

Begin
Notify IN_PROGRESS
Nwk discovery over bdbPrimaryChannelSet
Was formation successful?
Yes
No
Logical type Coordinator?
Yes
No
Nwk discovery over bdbSecondaryChannelSet
Is bdbSecondaryChannelSet = 0? Or has been already attempted??
No
Yes
Initiate Trust Center functionality
Notify FORMATION_FAILURE
Notify SUCCESS
End

Figure 18: Network formation.

15.7 Finding and Binding

The finding and binding procedure can be performed as initiator, target, or both, depending on the clusters that the endpoint performing the Finding and Binding procedure has. This means that if an endpoint has a cluster that is meant to be initiator, the finding and binding process for this endpoint will be executed as initiator. The definitions for initiator or target on clusters can be found in ZigBee ZCL specification [5].
The application must specify with which endpoint it wants to perform the finding and binding procedure by calling `bdb_SetIdentifyActiveEndpoint()`. Note that the endpoint indicated must contain the Identify cluster in order to be able to perform the procedure.

15.7.1 Finding & binding procedure for a target endpoint

When finding and binding is triggered on a target endpoint, the endpoint identifies itself for a finite period of time and handles the identify query commands from the initiator device. This commissioning mode will notify when it starts with a `BDB_COMMISSIONING_FB_INITIATOR_IN_PROGRESS` notification, but it will not notify when the commissioning process finishes by a BDB notification. Instead, the application must use `bdb_RegisterIdentifyTimeChangeCB()` to register a callback in which its handled the changes in the identify cluster.

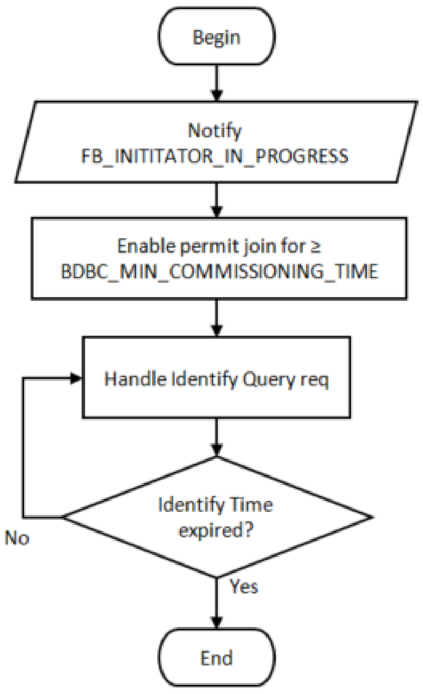

Begin
Notify
FB_INITITATOR_IN_PROGRESS
Enable permit join for ≥
BDBC_MIN_COMMISSIONING_TIME
Handle Identify Query req
Identify Time
expired?
No
Yes
End

Figure 19: Finding and binding procedure for a target endpoint.

15.7.2 Finding and binding procedure for an initiator endpoint

In this procedure, the initiator will search for identifying endpoints by sending identify query commands in broadcast message and requesting a simple descriptor for each node found. Then the binds for matching application clusters are created in the initiator. If group bind is requested, the initiator endpoint configures a group membership of target endpoint.

The finding and binding process for an initiator device is illustrated in Figure 20 and described here:

1. The application is notified about the commissioning method starting and the local device broadcast an Identify Query command.
    a. If no identify query responses are received over the process, then the application receives a `BDB_COMMISSIONING_FB NO_IDENTIFY_QUERY_RESPONSE` and the process finishes.
    b. If the device receives one or more responses then the device creates a list of the device responses.
2. The local device sends a ZDO simple descriptor request to each device in the list (one at the time), if no response is receive for the simple descriptor request the same device is retried up to `FINDING_AND_BINDING_MAX_ATTEMPTS` times after the other devices in the list are tried.
3. Upon the reception of a simple descriptor response, the local device will search for opposite matching application clusters with the endpoint in the local device that is performing the Finding and Binding procedure. For each matching cluster the local device will create a bind for the local device endpoint-cluster with the remote device from which it received the simple descriptor response. Refer to ZigBee ZCL specification for the definition of application clusters.
    a. The application can register a callback with `bdb_RegisterBindNotificationCB()` to receive notifications on each bind created by this or any other mean.
    b. If the bind table gets full during this process the application will receive a `BDB_COMMISSIONING_FB TABLE_FULL` notification and the process will be finished.
4. If `BDB_DEFAULT_COMMISSIONING_GROUP_ID` is defined as different from `0xFFFF`, then the remote device is notified to be part of the group.

5. The local device will repeat the steps from 2 to 4 until all the devices that did respond to the Identify Query command are attempted and then a `BDB_COMMISSIONING_SUCCESS` will be notified to the application.

The finding and binding process for an initiator device can be configured to be performed periodically every `FINDING_AND_BINDING_PERIODIC_TIME` seconds over `BDBC_MIN_COMMISSIONING_TIME` (180) seconds. This is defined by FINDING_AND_BINDING_PERIODIC_ENABLE, which by default is set to TRUE. In this case, if the same device responds to multiple identify query commands from the local device, this will be not be duplicated in the list and will be attempted up to `FINDING_AND_BINDING_MAX_ATTEMPTS` times.

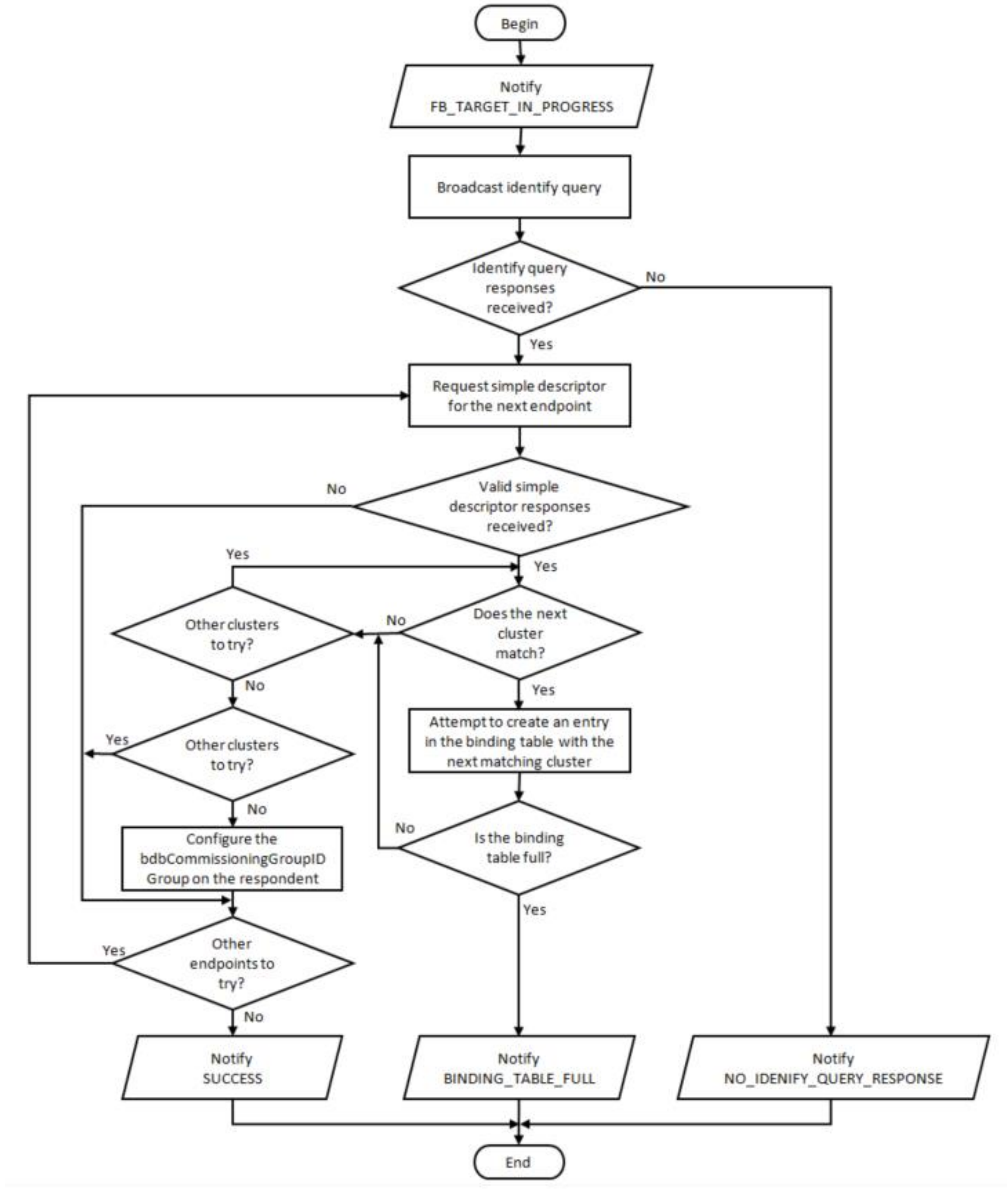

Begin
Notify FB_TARGET_IN_PROGRESS
Broadcast identify query
Identify query responses received?
No
Yes
Request simple descriptor for the next endpoint
Valid simple descriptor responses received?
No
Yes
Yes
Other clusters to try?
No
Does the next cluster match?
No
Yes
Yes
Other clusters to try?
No
Attempt to create an entry in the binding table with the next matching cluster
Configure the bdbCommissioningGroupID Group on the respondent
No
Is the binding table full?
Yes
Yes
Other endpoints to try?
No
Notify SUCCESS
Notify BINDING_TABLE_FULL
Notify NO_IDENIFY_QUERY_RESPONSE
End

Figure 20: Finding and binding procedure for an initiator endpoint.

15.8 Touchlink Commissioning

The Touchlink commissioning is an optional commissioning mechanism defined in the ZigBee BDB specification where nodes are commissioned using inter-PAN communication and requires device physical proximity.

15.8.1 Configurations

The following configurations must be modified by the user to create a valid Touchlink device. They are all found in the file *bdb_interface.h*:

15.8.1.1 Key Installation

All commercial Touchlink products use the "Touchlink master key" and the "Touchlink pre-installed link key" set. This set of keys could be available to manufacturers which have a successfully certified a Touchlink product, using the certification keys set provided by default.

Note that any Touchlink implementation will not be able to interoperate with commercial Touchlink devices without the Touchlink master keys. Once the Touchlink master keys have been achieved, they should be installed in the code with the following modifications:

1. Overwrite the `TOUCHLINK_CERTIFICATION_ENC_KEY` `TOUCHLINK_CERTIFICATION_LINK_KEY` and with the actual secret values.
2. Change the `TOUCHLINK_KEY_INDEX` definition to `TOUCHLINK_KEY_INDEX_MASTER`.

15.8.1.2 Constants

The BDB defines constants and internal attributes defaults to allow a device to manage the way the Touchlink device operates (see section 5.2 in Base Device Behavior Specification [7]).

Definition	**Specification's Constant / Attribute default**	**Value**
`BDBCTL_INTER_PAN_TRANS_ID_LIFETIME`	*bdbcTLInterPANTransIdLifetime*	8000
`BDBCTL_MIN_STARTUP_DELAY_TIME`	*bdbcTLMinStartupDelayTime*	2000
`BDBCTL_PRIMARY_CHANNEL_LIST`	*bdbcTLPrimaryChannelSet*	0x02108800
`BDBCTL_RX_WINDOW_DURATION`	*bdbcTLRxWindowDuration*	5000
`BDBCTL_SCAN_TIME_BASE_DURATION`	*bdbcTLScanTimeBaseDuration*	250
`BDBCTL_SECONDARY_CHANNEL_LIST`	*bdbcTLSecondaryChannelSet*	0x05EF7000

Table 2: Definitions Derived From the ZigBee Base Device Behavior Specification

15.8.1.3 Endpoint Setup

Since the Touchlink commissioning is managed by a dedicated task separated from the applications, its endpoint could be re-defined to any valid value, which is not in use by any application endpoint on the device, and its device ID, appearing in the simple descriptor, should not be equal to any valid value to prevent accidental match.

`TOUCHLINK_INTERNAL_ENDPOINT` (default = 13).
`TOUCHLINK_INTERNAL_DEVICE_ID` (default = 0xE15E).

15.8.1.4 Identify Sequence Time Interval

In the Touchlink commissioning sequence, if an appropriate scan response command is received, the initiator will send an Identify command to the chosen target and then wait for a time interval defined by the following parameter (in milliseconds) before sending a network start or network join command.

When an Identify Request command is received with identify duration field value set to 0xffff (default time known by the receiver), the application's Identify callback function will be called with a duration value set according to the following parameter (in seconds):

`TOUCHLINK_DEFAULT_IDENTIFY_TIME` – identify duration if not specified in the received command (default = 3).

It is possible to gracefully abort a touch-link process (see section 8.7 in Base Device Behavior Specification 2), until the end of this time interval. Beyond that, target state may change irreversibly. If abort is employed and controlled by a human interaction, it is recommended to increase this value (e.g. to 2000). Please note that increasing it to a higher value than the default also increases the risk of touch-link failure due to transaction lifetime expiration, especially if done on the secondary channel set.

15.8.1.5 Free Ranges Split Thresholds

When initiating Touchlink commissioning with devices which are capable of assigning addresses, ranges of free network addresses and group identifiers held by the initiator could be split and passed to the target. The initiator can split its ranges as long as the remaining range and the target range are no less than the minimum size, defined by the following parameters:

`TOUCHLINK_ADDR_THRESHOLD` – the minimum size of addresses range after split (default = 10).
`TOUCHLINK_GRP_ID_THRESHOLD` – the minimum size of group identifiers range after split (default = 10).

15.8.1.6 Free Ranges Split Thresholds

This feature allows overriding the default RSSI-based target selection during Touchlink, with an application-specific selection function. An application selection function could be used in scenarios where multiple target are expected to have similar RSSI (e.g. multiple lights bundled together), and allows integrating other parameters in the selection (e.g. Factory New state, previously selected device, etc.).

15.8.2 Development-Only Parameters

The following parameters, if enabled, will break Touchlink conformity and security rules. They may be used to assist during development, but must be disabled before release. All the parameters could be uncommented in *bdb.h* file, instead of being defined globally in the project.

15.8.2.1 Channel Offset

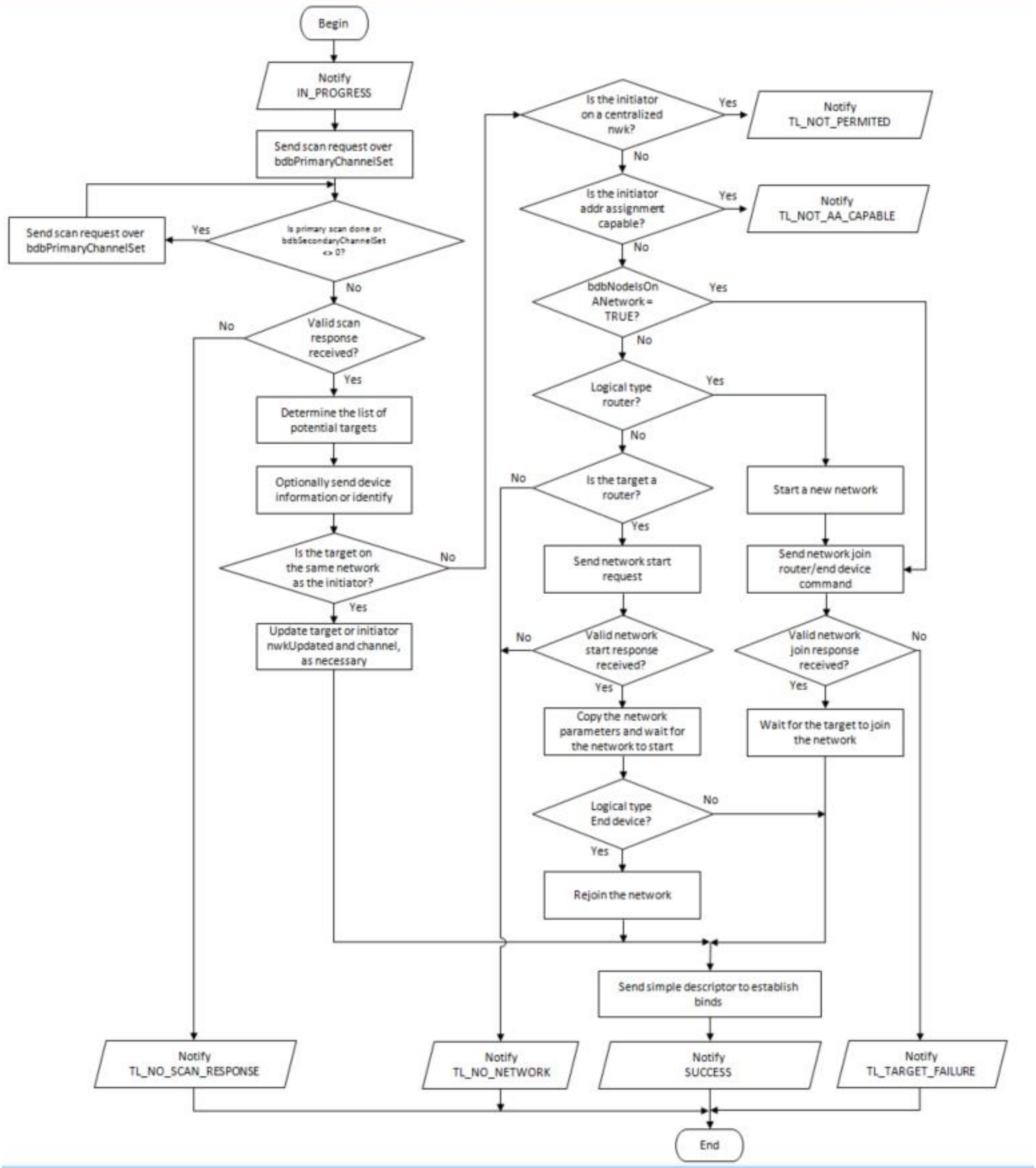
The flags `Ch_Plus_1`, `Ch_Plus_2`, or `Ch_Plus_3`, can be set in the TOUCHLINK_CH_OFFSET definition in bdb.h to shift the primary channel set, allowing testing of multiple Touchlink devices set in the same space without interference only for testing purposes. TOUCHLINK_CH_OFFSET is defined by default as No_Ch_offset, this means that no shift is applied to primary channel set.

15.8.2.2 Fixed First Channel Selection

The flag `TOUCHLINK_DEV_SELECT_FIRST_CHANNEL`, if enabled during compilation, will override the random channel selection mechanism employed by the Touchlink device, and will set it to always select the first primary channel.

15.8.3 Touchlink commissioning procedure for an initiator

In this procedure the initiator scans for nodes that support touchlink, and if any are found, the touchlink commissioning procedure establishes a new distributed network with the target.

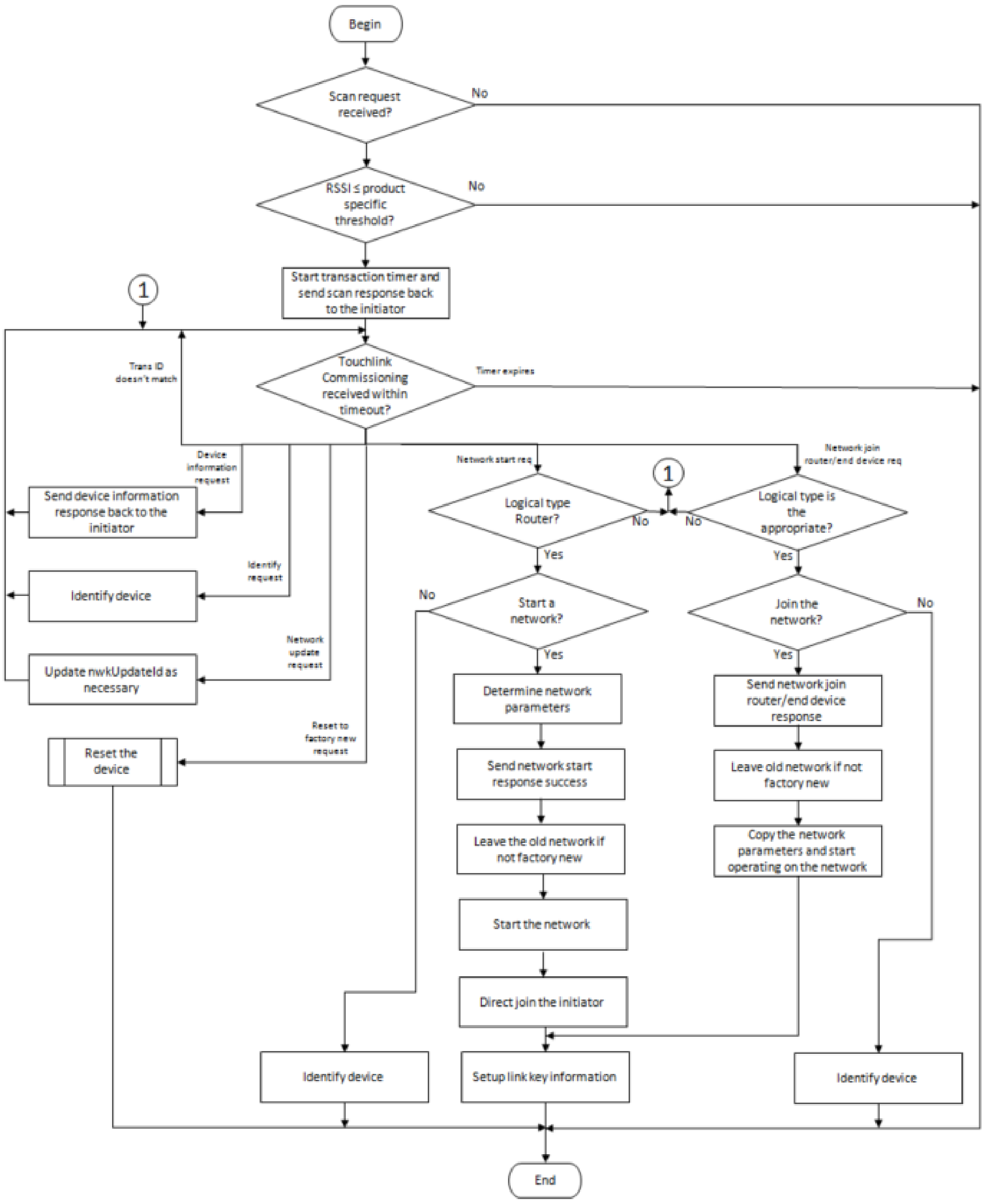


Begin
Notify IN_PROGRESS
Send scan request over bdbPrimaryChannelSet
Is primary scan done or bdbSecondaryChannelSet <> 0?
Yes
Send scan request over bdbPrimaryChannelSet
No
Valid scan response received?
No
Yes
Determine the list of potential targets
Optionally send device information or identify
Is the target on the same network as the initiator?
No
Yes
Update target or initiator nwkUpdated and channel, as necessary
Is the initiator on a centralized nwk?
Yes
Notify TL_NOT_PERMITED
No
Is the initiator addr assignment capable?
Yes
Notify TL_NOT_AA_CAPABLE
No
bdbNodeIsOnANetwork = TRUE?
Yes
No
Logical type router?
Yes
No
Start a new network
Is the target a router?
No
Yes
Send network start request
Send network join router/end device command
Valid network start response received?
No
Yes
Valid network join response received?
No
Yes
Copy the network parameters and wait for the network to start
Wait for the target to join the network
Logical type End device?
No
Yes
Rejoin the network
Send simple descriptor to establish binds
Notify TL_NO_SCAN_RESPONSE
Notify TL_NO_NETWORK
Notify SUCCESS
Notify TL_TARGET_FAILURE
End

Figure 21: Touchlink commissioning procedure for an initiator.

15.8.4 Touchlink commissioning procedure for a target

In this procedure, the target responds to touchlink commissioning commands from the initiator to start a new network or join to the initiator network.



Begin
Scan request received?
No
RSSI ≤ product specific threshold?
No
Start transaction timer and send scan response back to the initiator
1
Trans ID doesn't match
Touchlink Commissioning received within timeout?
Timer expires
Device information request
Send device information response back to the initiator
Identify request
Identify device
Network update request
Update nwkUpdateId as necessary
Reset to factory new request
Reset the device
Network start req
Logical type Router?
No
1
No
Network join router/end device req
Logical type is the appropriate?
Yes
Yes
No
Start a network?
Yes
Determine network parameters
Send network start response success
Leave the old network if not factory new
Start the network
Direct join the initiator
Join the network?
No
Yes
Send network join router/end device response
Leave old network if not factory new
Copy the network parameters and start operating on the network
Identify device
Setup link key information
Identify device
End

Figure 22: Touchlink commissioning procedure for a target.

15.9 Reset procedures

Base device behavior defines how the device must act upon reception of reset commands over-the-air or by user interaction as follows:

15.9.1 Reset via Basic Cluster

If the reception of this is supported, then the application must reset the all the attributes in all clusters supported by the device. No network parameters, binds or groups must be affected by this command. This must be implemented by the application in the callback for this command.

15.9.2 Reset via Touchlink Commissioning Cluster

If touchlink as target is supported, this reset mechanism will trigger the device to process a leave request to itself with *Rejoin* set to FALSE and *RemoveChildren* set to FALSE. See section 15.9.5 for further details on process of leave request.

15.9.3 Reset via Mgmt_leave_req ZDO command

If the command is valid, the receiving device will process a leave request for itself with *Rejoin* set to FALSE and *RemoveChildren* set to FALSE. See section 15.9.5 for further details on process of leave request.

15.9.4 Reset via Local Action

This type of reset is the one that the user will trigger when pressing a special button or perform a sequence to reset to factory new the local device. This is processed as a network leave request to itself with *Rejoin* set to FALSE and *RemoveChildren* set to FALSE for non-Coordinator devices. For coordinator devices, this implements a sequence of steps to clear the same ZigBee persistence data as a network leave command would do, as coordinator devices cannot process network leave commands. See section 15.9.5 for further details on process of leave request. To perform this action call the function `bdb_resetLocalAction()`.

15.9.5 Reset via Nwk Leave request

Network leave request is processed depending on the logical device that receives the command. Coordinator devices ignore the command (including those issued from itself), while End Devices only accept the command if issued by itself or its parent device. Router devices process the command from itself and from any other device in the network if *zgNwkLeaveRequestAllowed* is enabled. A valid request will cause the device to clear all persistent ZigBee data (Bindings, network parameters, groups, attributes, etc.) except for the outgoing network frame counter for the network that is being left.

16. Network Manager

16.1 Overview

A single device can become the Network Manager. This device acts as the central mechanism for reception of network:

- Channel Interference reports and changing the channel of the network if interference is detected, and
- PAN ID Conflict reports and changing the PAN ID of the network if conflict is detected.

The default address of the Network Manager is the coordinator. However, this can be updated by sending a *Mgmt_NWK_Update_req* command with a different short address for the Network Manager. The device that is the Network Manager sets the network manager bit in the server mask in the node descriptor and responds to *System_Server_Discovery_req* commands.

The Network Manager implementation resides in *ZDNwkMgr.c* and *ZDNwkMgr.h* files.

16.2 Channel Interference

The Network Manager implements frequency agility measures in the face of interference. This section explains how, through the use of the *Mgmt_NWK_Update_req* and *Mgmt_NWK_Update_notify* commands, the channel of a network can be changed.

16.2.1 Channel Interference Detection

Each router or coordinator tracks transmit failures using the Transmit Failure field in the neighbor table and also keeping a NIB counter for Total Transmissions attempted. Once the total transmissions attempted is over `ZDNWKMGR_MIN_TRANSMISSIONS` (20), if the transmit failures exceeds `ZDNWKMGR_CI_TX_FAILURE` (25) percent of the messages sent, the device may have detected interference on the channel in use.

The device then takes the following steps:

1. Conduct an energy scan on all channels. If this energy scan does not indicate higher energy on the current channel than other channels, no action is taken. The device should continue to operate as normal and the message counters are not reset.
2. If the energy scan does indicate increased energy on the channel in use, a *Mgmt_NWK_Update_notify* should be sent to the Network Manager to indicate interference is present. This report is sent as an APS unicast with acknowledgement and once the acknowledgment is received the total transmit and transmit failure counters are reset to zero.
3. To avoid a device with communication problems from constantly sending reports to the Network Manager, the device does not send a *Mgmt_NWK_Update_notify* more than 4 times per hour.

16.2.2 Channel Interference Resolution

Upon receipt of an unsolicited *Mgmt_NWK_Update_notify*, the Network Manager applies different methods to best determine when a channel change is required and how to select the most appropriate channel.

The Network Manger does the following:

1. Upon receipt of the *Mgmt_NWK_Update_notify*, the Network Manager determines if a channel change is required using the following criteria:
    a. If any single device has more than `ZDNWKMGR_CC_TX_FAILURE` (50) percent transmission failures a channel change should be considered.
    b. The Network Manager compares the failure rate reported on the current channel against the stored failure rate from the last channel change. If the current failure rate is higher than the last failure rate then the channel change is considered.
2. If the above data indicate a channel change should be considered, the Network Manager completes the following:

   a. Select a single channel based on the *Mgmt_NWK_Update_notify* based on the lowest energy. This is the proposed new channel. If this new channel does not have an energy level below an acceptable threshold `ZDNWKMGR_ACCEPTABLE_ENERGY_LEVEL`, a channel change should not be done.

3. Prior to changing channels, the Network Manager stores the energy scan value as the last energy scan value and the failure rate from the existing channel as the last failure rate.
4. The Network Manager broadcasts (to all routers and coordinator) a *Mgmt_NWK_Update_req* notifying devices of the new channel. It then increments the *nwkUpdateId* parameter in the NIB and beacon payload, and includes it in the *Mgmt_NWK_Update_req*. The Network Manager sets a timer based on the value of `ZDNWKMGR_UPDATE_REQUEST_TIMER` (i.e., *apsChannelTimer*) upon issue of a *Mgmt_NWK_Update_req* that changes channels and will not issue another such command until this timer expires.
5. Upon issue of a *Mgmt_NWK_Update_req* with a change of channels, the local Network Manager sets a timer equal to the *nwkNetworkBroadcastDeliveryTime* and switches channels upon expiration of this timer.

Upon receipt of a *Mgmt_NWK_Update_req* with a change of channels from the Network Manager, a device sets a timer equal to the *nwkNetworkBroadcastDeliveryTime* and switches channels upon expiration of this timer. Each node stores the received *nwkUpdateId* in the NIB and beacon payload, and also resets the total transmit count and the transmit failure counters.

For devices with *RxOnWhenIdle* equals FALSE, any network channel change will not be received. On these devices or routers that have lost the network, an active scan is conducted on the *channelList* in the NIB (i.e., *apsChannelMask*) using the extended PAN ID (EPID) to find the network. If the extended PAN ID is found on different channels, the device selects the channel with the higher value in the *nwkUpdateId* parameter. If the extended PAN ID is not found using the *apsChannelMask* list, a scan is completed using all channels.

16.2.3 Quick Reference

Setting minimum transmissions attempted for Channel Interference detection	Set `ZDNWKMGR_MIN_TRANSMISSIONS` (in `ZDNwkMgr.h`)
Setting minimum transmit failure rate for Channel Interference detection	Set `ZDNWKMGR_CI_TX_FAILURE` (in `ZDNwkMgr.h`)
Setting minimum transmit failure rate for Channel Change	Set `ZDNWKMGR_CC_TX_FAILURE` (in `ZDNwkMgr.h`)
Setting acceptable energy level threshold for Channel Change	Set `ZDNWKMGR_ACCEPTABLE_ENERGY_LEVEL` (in `ZDNwkMgr.h`)
Setting APS channel timer for issuing Channel Changes	Set `ZDNWKMGR_UPDATE_REQUEST_TIMER` (in `ZDNwkMgr.h`)

16.3 PAN ID Conflict

Since the 16-bit PAN ID is not a unique number there is a possibility of a PAN ID conflict in the local neighborhood. The Network Manager implements PAN ID conflict resolution. This section explains how, through the use of the Network Report and Update commands, the PAN ID of a network can be updated.

16.3.1 PAN ID Conflict Detection

Any device that is operational on a network and receives a beacon in which the PAN ID of the beacon matches its own PAN ID but the EPID value contained in the beacon payload is either not present or not equal to *nwkExtendedPANID*, is considered to have detected a PAN ID conflict.

A node that has detected a PAN ID conflict sends a Network Report command of type PAN ID conflict to the designated Network Manager identified by the *nwkManagerAddr* in the NIB. The Report Information field will contain a list of all the 16-bit PAN identifiers that are being used in the local neighborhood. The list is constructed from the results of an ACTIVE scan.

16.3.2 PAN ID Conflict Resolution

On receipt of the Network Report command, the Network Manager selects a new 16-bit PAN ID for the network. The new PAN ID is chosen at random, but a check is performed to ensure that the chosen PAN ID is not contained within the Report Information field of the network report command.

Once a new PAN ID has been selected, the Network Manager first increments the NIB attribute *nwkUpdateID* and then constructs a Network Update command of type PAN identifier update. The Update Information field is set to the value of the new PAN ID. After it sends out this command, the Network Manager starts a timer with a value equal to *nwkNetworkBroadcastDeliveryTime* seconds. When the timer expires, it changes its current PAN ID to the newly selected one.

On receipt of a Network Update command of type PAN ID update from the Network Manager, a device (in the same network) starts a timer with a value equal to *nwkNetworkBroadcastDeliveryTime* seconds. When the timer expires, the device changes its current PAN ID to the value contained within the Update Information field. It also stores the new received *nwkUpdateID* in the NIB and beacon payload.

17. Green Power

17.1 Introduction

As a requirement for Z3.0 certification, all ZigBee routing devices (coordinators, routers) must support the Green Power Basic proxy, which is an application that can relay commands from a GPD to a GP Sink device.
A GPD is a device that has a very limited power or relies on energy harvesting for functioning, and it cannot perform the two ways communication to establish association to a ZigBee network. These GPDs use Inter-PAN frames to commission itself into the network or to deliver commands. The commissioning methods and the type of commands supported by the GPD will depend on its capabilities and resources. The details of those commissioning methods and commands are out of the scope of this document.
The Basic proxy requires the implementation of GP stub and GP cluster. The GP stub handles the Inter-PAN commands and passes those to the GP endpoint application. It also sends GP data frames back to the GPD for certain commissioning methods. The GP stub defined in a way that different applications can sit on the top of it, such as Sink Device. The implementation of a Sink device is out of the scope of this document. For further details on Sink Device implementation refer to [8], also for the GP stub interface, refer to [1] and the header file *dGP_stub.h*.
GP is implemented in the ZigBee reserved endpoint 242.

17.2 Green Power Basic Proxy

Since the GP basic proxy is an application to relay the commands to a Sink device, it does not provides a functionality that needs to be handled by the actual application running in the basic proxy device implementing it (this means your actual application, Light, Switch, etc.). The only interface that this functionality has is the following:

- `gp_RegisterGPChangeChannelReqCB()`: Register a callback to ask to your application for permission to switch the operational channel to the GPD's channel to perform the commissioning of the GPD during at most `gpBirectionalCommissioningChangeChannelTimeout` (5 seconds). The callback registered can return FALSE to not allow the change of the channel if an application operation cannot be interrupted. BDB module is also check for operations before asking the application. If the application returns TRUE or no callback is registered the GP basic proxy application will handle the change of channels.

18. Inter-PAN Transmission

18.1 Overview

Inter-PAN transmission enables ZigBee devices to perform limited, insecure, and possibly anonymous exchange of information with devices in their local neighborhood without having to form or join the same ZigBee network.

The Inter-PAN feature is implemented by the Stub APS layer, which can be included in a project by defining the `INTER_PAN` compile option and including *stub_aps.c* and *stub_aps.h* files in the project.

18.2 Data Exchange

Inter-PAN data exchanges are handled by the Stub APS layer, which is accessible through INTERP-SAP, parallel to the normal APSDE-SAP:

- The `INTERP_DataReq()` and `APSDE_DataReq()` are invoked from `AF_DataRequest()` to send Inter-PAN and Intra-PAN messages respectively.
- The `INTERP_DataIndication()` invokes `APSDE_DataIndication()` to indicate the transfer of Inter-PAN data to the local application layer entity. The application then receives Inter-PAN data as a normal incoming data message (`APS_INCOMING_MSG`) from the APS sub-layer with the source address belonging to an external PAN (verifiable by `StubAPS_InterPan()` API) .
- The `INTERP_DataConfirm()` invokes `afDataConfirm()` to send an Inter-PAN data confirm back to the application. The application receives a normal data confirm (`AF_DATA_CONFIRM_CMD`) from the AF sub-layer.

The Stub APS layer also provides interfaces to switch channel for Inter-PAN communication and check for Inter-PAN messages. Please refer to the Z-Stack API [1] document for detailed description of the Inter-PAN APIs.

The `StubAPS_InterPan()` API is used to check for Inter-PAN messages. A message is considered as an Inter-PAN message if it meets one the following criteria:

- The current communication channel is different that the device's NIB channel, or
- The current communication channel is the same as the device's NIB channel *but* the message is destined for a PAN different than the device's NIB PAN ID, or
- The current communication channel is the same as the device's NIB channel *and* the message is destined for the same PAN as device's NIB PAN ID *but* the destination application endpoint is an Inter-PAN endpoint (`0xFE`). This case is true for an Inter-PAN response message that's being sent back to a requestor.

A typical usage scenario for Inter-PAN communication is as follows. The initiator device:-

- Calls `StubAPS_AppRegister()` API to register itself with the Stub APS layer
- Calls `StubAPS_SetInterPanChannel()` API to switch its communication channel to the channel in use by the remote device
- Specifies the destination PAN ID and address for the Inter-PAN message about to be transmitted
- Calls `AF_DataRequest()` API to send the message to the remote device through Inter-PAN channel
- Receives back (if required) a message from the remote device that implements the Stub APS layer and is able to respond
- Calls `StubAPS_SetIntraPanChannel()` API to switch its communication channel back to its original channel

18.2.1 Quick Reference

Setup application as InterPAN application.	Call `StubAPS_RegisterApp( app_endpoint )`
Set InterPAN channel.	Call `StubAPS_SetInterPanChannel( channel )`
Send InterPAN Message.	Call `AF_DataRequest()` with: • dstPanID different from `_NIB.nwkPanId` • dst address endpoint == `STUBAPS_INTER_PAN_EP`
Receive an InterPAN message.	Receive an `OSAL AF_INCOMING_MSG_CMD` message with an incoming DstEndPoint == `STUBAPS_INTER_PAN_EP`
End the InterPAN session by putting back the IntraPAN channel.	Call `StubAPS_SetIntraPanChannel()`

19. ZMAC LQI Adjustment

19.1 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` provides capability to adjust the default TI MAC value of LQI by taking the correlation into account.

19.2 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().`

19.3 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 [1] 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.

20. Heap Memory Management

20.1 Overview

The OSAL heap memory manager provides a POSIX-like API for allocating and re-cycling dynamic heap memory. Two important considerations in a low-cost, resource-constrained embedded system, size and speed, have been duly addressed in the implementation of the heap memory manager.

- Overhead memory cost to manage each allocated block has been minimized – as little as *2 bytes* on CPU's with one- or two-byte-aligned memory access (e.g. 8051 SOC and MSP430).

- Interrupt latency for the allocation and free operations has been minimized – freeing is immediate with no computational load other than bounds checks and clearing a bit; allocating is very much sped-up with a packed long-lived memory block and a dynamically updated first-free pointer for high-frequency small-block allocations (e.g. OSAL Timers).

20.2 API

20.2.1 osal_mem_alloc()

The `osal_mem_alloc()` function is a request to the memory manager to reserve a block of the heap.

20.2.1.1 Prototype

```
void *osal_mem_alloc( uint16 size );
```

20.2.1.2 Parameters

`size` – the number of bytes of dynamic memory requested.

20.2.1.3 Return

If a big enough free block is found, the function returns a void pointer to the RAM location of the heap memory reserved for use. A NULL pointer is returned if there isn't enough memory to allocate. Any non-NULL pointer returned must be freed for re-use by invoking `osal_mem_free()`.

20.2.2 osal_mem_free()

The `osal_mem_free()` function is a request to the memory manager to release a previously reserved block of the heap so that the memory can be re-used.

20.2.2.1 Prototype

```
void osal_mem_free( void *ptr );
```

20.2.2.2 Parameters

`ptr` – a pointer to the buffer to release for re-use – this pointer must be the non-NULL pointer that was returned by a previous call to osal_mem_alloc().

20.2.2.3 Return

None.

20.3 Strategy

Memory management should strive to maintain contiguous free space in the heap, in as few blocks as possible, with each block as big as possible. Such a general strategy helps to ensure that requests for large memory blocks always succeed if the total heap size has been set properly for the application's use pattern.

The following specific strategies have been implemented:

- Memory allocation is not penalized by having to traverse long-lived heap allocations if the system initialization is implemented as recommended within this guide.
- Memory allocation for small-blocks almost always begins searching at the first free block in the heap.
- Memory allocation attempts to coalesce all contiguous free blocks traversed in an attempt to form a single free block large enough for an allocation request.
- Memory allocation uses the first free block encountered (or created by coalescing) that is big enough to meet the request; the memory block is split if it is usefully bigger than the requested allocation.

20.4 Discussion

It is immediately after system task initialization that the effective "start of the heap" mark is set to be the first free block. Since the memory manager always starts a "walk", looking for a large enough free block, from the aforementioned mark, it will **greatly** reduce the run-time overhead of the walk if all long-lived heap allocations are packed at the start of the heap so that they will not have to be traversed on every memory allocation. Therefore, any application should make all long-lived dynamic memory allocations in its respective system initialization routine (e.g. `XXX_Init()`, where `XXX` is the Application Name). Within said system initialization routines, the long-lived items must be allocated before any short-lived items. Any short-lived items allocated must be freed before returning, otherwise the long-lived bucket may be fragmented and the run-time throughput adversely affected proportionally to the number of long-lived items that the OSAL_Memory module is forced to iterate over *for every allocation* for the rest of the life of the system. As an example, if the system initialization function starts an OSAL Timer (`osal_start_timerEx()`), this may fragment the long-lived bucket because the memory allocated for the timer will be freed and re-used throughout the life of the system (even if coincidence happens that every free and re-use is simply for resetting the same timer.) The recommended solution in this case would be to set the event corresponding to the timer (`osal_set_event ()`) and then continue to restart the timer as appropriate in the application's event handle for the corresponding event (refer to the behavior of the hal_key polling timer and corresponding event, `HAL_KEY_EVENT`). On the other hand, a reload timer (`osal_start_reload_timer()`) is a long-lived allocation and is recommended to be started during system initialization of all other long-lived items.

The application implementer must ensure that their use of dynamic memory does not adversely affect the operation of the underlying layers of the Z-Stack. The Z-Stack is tested and qualified with sample applications that make minimal use of heap memory. Thus, the user application that uses significantly more heap than the sample applications, or the user application that is built with a smaller value set for `MAXMEMHEAP` than is set in the sample applications, may inadvertently starve the lower layers of the Z-Stack to the point that they cannot function effectively or at all. For example, an application could allocate so much dynamic memory that the underlying layers of the stack would be unable to allocate enough memory to send and/or receive any OTA messages – the device would not be seen to be participating OTA.

20.5 Configuration

20.5.1 MAXMEMHEAP

The `MAXMEMHEAP` constant is usually defined in `OnBoard.h`. It must be defined to be less than **32768**.

`MAXMEMHEAP` is the number of bytes of RAM that the memory manager will reserve for the heap – it cannot be changed dynamically at runtime – it must be defined at compile-time. If `MAXMEMHEAP` is defined to be greater than or equal to `32768`, a compiler error in `OSAL_Memory.c` will trigger. `MAXMEMHEAP` does not reflect the total amount of dynamic memory that the user can expect to be usable because of the overhead cost per memory allocation.

20.5.2 OSALMEM_PROFILER

The `OSALMEM_PROFILER` constant is defined locally in `OSAL_Memory.c` to be `FALSE` by default.

After the implementation of a user application is mature, the OSAL memory manager may need to be re-tuned in order to achieve optimal run-time performance with regard to the `MAXMEMHEAP` and `OSALMEM_SMALL_BLKSZ` constants defined. The code enabled by defining the `OSALMEM_PROFILER` constant to `TRUE` allows the user to gather the empirical, run-time results required to tune the memory manager for the application. The profiling code does the following.

20.5.2.1 OSALMEM_INIT

The `OSALMEM_INIT` constant is defined locally in `OSAL_Memory.c` to be ascii '**X**'.

The memory manager initialization sets all of the bytes in the heap to the value of `OSALMEM_INIT`.

20.5.2.2 OSALMEM_ALOC

The `OSALMEM_ALOC` constant is defined locally in `OSAL_Memory.c` to be ascii '**A**'.

The user available bytes of any block allocated are set to the value of `OSALMEM_ALOC`.

20.5.2.3 OSALMEM_REIN

The `OSALMEM_REIN` constant is defined locally in `OSAL_Memory.c` to be ascii '**F**'.

Whenever a block is freed, what had been the user available bytes are set to the value of `OSALMEM_REIN`.

20.5.2.4 OSALMEM_PROMAX

The `OSALMEM_PROMAX` constant is defined locally in `OSAL_Memory.c` to be **8**.

`OSALMEM_PROMAX` is the number of different bucket sizes to profile. The bucket sizes are defined by an array:

```
  static uint16 proCnt[OSALMEM_PROMAX] = { OSALMEM_SMALL_BLKSZ,
                                       48, 112, 176, 192, 224, 256, 65535 };
```

The bucket sizes profiled should be set according to the application being tuned, but the last bucket must always be 65535 as a catch-all. There are 3 metrics kept for each bucket.

- `proCur` – the current number of allocated blocks that fit in the corresponding bucket size.

- `proMax` – the maximum number of allocated blocks that corresponded to the bucket size at once.
- `proTot` – the total number of times that a block was allocated that corresponded to the bucket size.

In addition, there is a count kept of the total number of times that the part of heap reserved for "small blocks" was too full to allow a requested small-block allocation: `proSmallBlkMiss`.

20.5.3 OSALMEM_MIN_BLKSZ

The `OSALMEM_MIN_BLKSZ` constant is defined locally in `OSAL_Memory.c`.

`OSALMEM_MIN_BLKSZ` is the minimum size in bytes of a block that is created by splitting a free block into two new blocks. The 1st new block is the size that is being requested in a memory allocation and it will be marked as in use. The 2nd block is whatever size is leftover and it will be marked as free. A larger number may result in significantly faster overall runtime of an application without necessitating any more or not very much more overall heap size. For example, if an application made a very large number of inter-mixed, short-lived memory allocations of 2 & 4 bytes each, the corresponding blocks would be 4 & 6 bytes each with overhead. The memory manager could spend a lot of time thrashing, as it were, repeatedly splitting and coalescing the same general area of the heap in order to accommodate the inter-mixed size requests.

20.5.4 OSALMEM_SMALL_BLKSZ

The `OSALMEM_SMALL_BLKSZ` constant is defined locally in `OSAL_Memory.c`.

The heap memory use of the Z-Stack was profiled using the GenericApp Sample Application and it was empirically determined that the best worst-case average combined time for a memory allocation and free, during a heavy OTA load, can be achieved by splitting the free heap into two sections. The first section is reserved for allocations of smaller-sized blocks and the second section is used for larger-sized allocations as well as for smaller-sized allocations if and when the first section is full. `OSALMEM_SMALL_BLKSZ` is the maximum block size in bytes that can be allocated from the first section.

20.5.5 OSALMEM_SMALLBLK_BUCKET

The `OSALMEM_SMALLBLK_BUCKET` constant is locally defined in `OSAL_Memory.c`.

`OSALMEM_SMALLBLK_BUCKET` is the number of bytes dedicated to the previously described first section of the heap which is reserved for smaller-sized blocks.

20.5.6 OSALMEM_NODEBUG

The `OSALMEM_NODEBUG` constant is locally defined in `OSAL_Memory.c` to be `TRUE` by default.

The Z-Stack and Sample Applications do not misuse the heap memory API. The onus to be equally correct is on the user application: in order to provide the minimum throughput latency possible, there are no run-time checks for correct use of the API. An application can be shown to be correct by defining the `OSALMEM_NODEBUG` constant to `FALSE`. Such a setting will enable code that traps on the following misuse scenario.

- Invoking `osal_mem_alloc()` with size equal to zero.

Warning: invoking `osal_mem_free()` with a dangling or invalid pointer cannot be detected.

20.5.7 OSALMEM_PROFILER_LL

The `OSALMEM_PROFILER_LL` constant is defined locally in `OSAL_Memory.c` to be `FALSE` by default.

Normally, the allocations that are packed into the Long-Lived bucket by all of the system initialization should not be counted during "profiling" because they are not iterated over during run-time. But, in order to properly tune the size of the Long-Lived bucket for any given Application, this constant should be used for one run on the debugger with a mature implementation. The numbers used in the following example are for the 8051 SOC, GenericApp, with out-of-the-box settings and thus using this default:

```
#define OSALMEM_LL_BLKSZ          (OSALMEM_ROUND(417) + (19 * OSALMEM_HDRSZ))
```

1. Define `OSALMEM_PROFILER` and `OSALMEM_PROFILER_LL` to `TRUE`
2. Set a break point in `osal_mem_kick()` after this operation:
    a. ff1 = tmp – 1;
3. Inspect the variable `proCur` in an IAR 'Watch' window and sum the counts of all of the buckets (19 in this example) and plug it into the formula above – this is the count of long-lived items.
4. Subtract the value of ff1 (0x1095 in one particular run) from the location of theHeap (0x0ECE in that same run) and then subtract the sub-total of the count of long-lived items multiplied by the `OSALMEM_HDRSZ` (19 * 2 = 38 for this example.)

Further memory profiling should now be done with `OSALMEM_PROFILER_LL` set back to `FALSE` so as not to count the long-lived allocations in the statistics.

21. Compile Options

21.1 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.

21.2 Requirements

21.2.1 Target Development System Requirements

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

Platform/Target	**Compiler/Tool**
SmartRF05EB + CC2530	IAR EW8051
SmartRF06EB + CC2538	IAR EWARM

21.3 Using Z-Stack Compile Options

21.3.1 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 (Coordinator, Router, End device). 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.

21.3.1.1 Compile Options In Linker Control Files

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

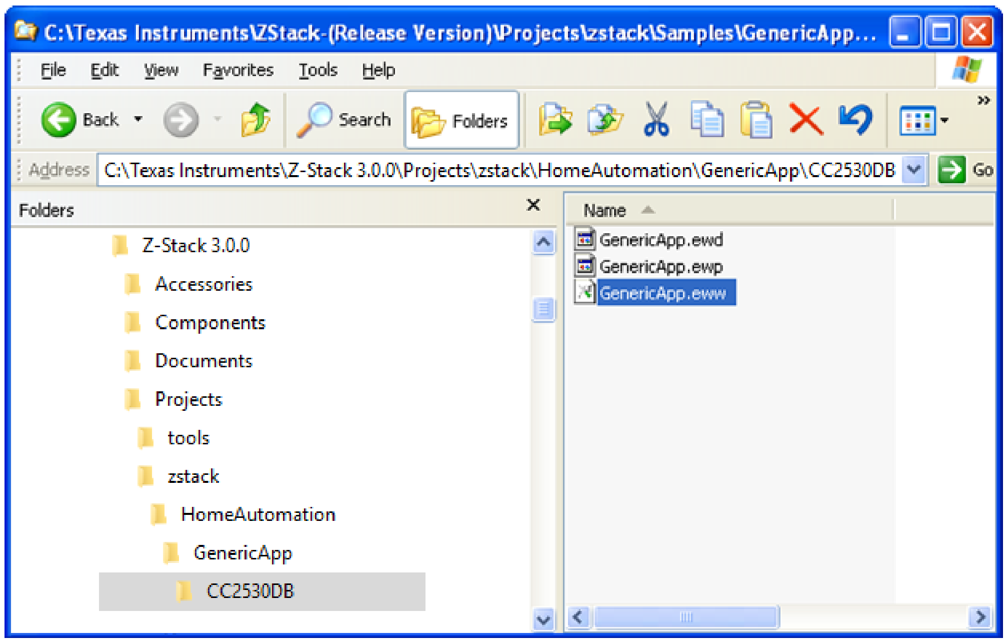
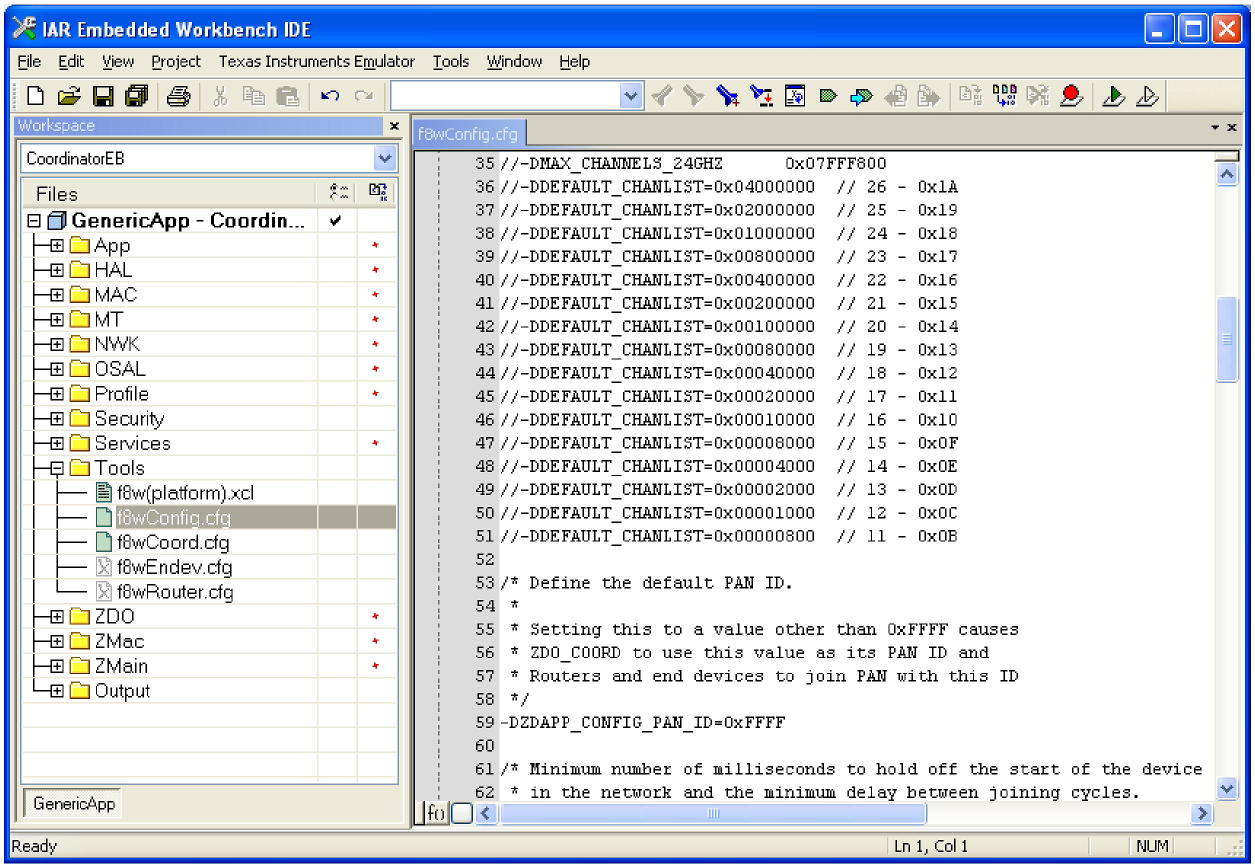

C:\Texas Instruments\ZStack-(Release Version)\Projects\zstack\Samples\GenericApp...
File Edit View Favorites Tools Help
Back
Search
Folders
Address C:\Texas Instruments\Z-Stack 3.0.0\Projects\zstack\HomeAutomation\GenericApp\CC2530DB
Go
Folders
Z-Stack 3.0.0
Accessories
Components
Documents
Projects
tools
zstack
HomeAutomation
GenericApp
CC2530DB
Name
GenericApp.ewd
GenericApp.ewp
GenericApp.eww

Figure 23: GenericApp project files location

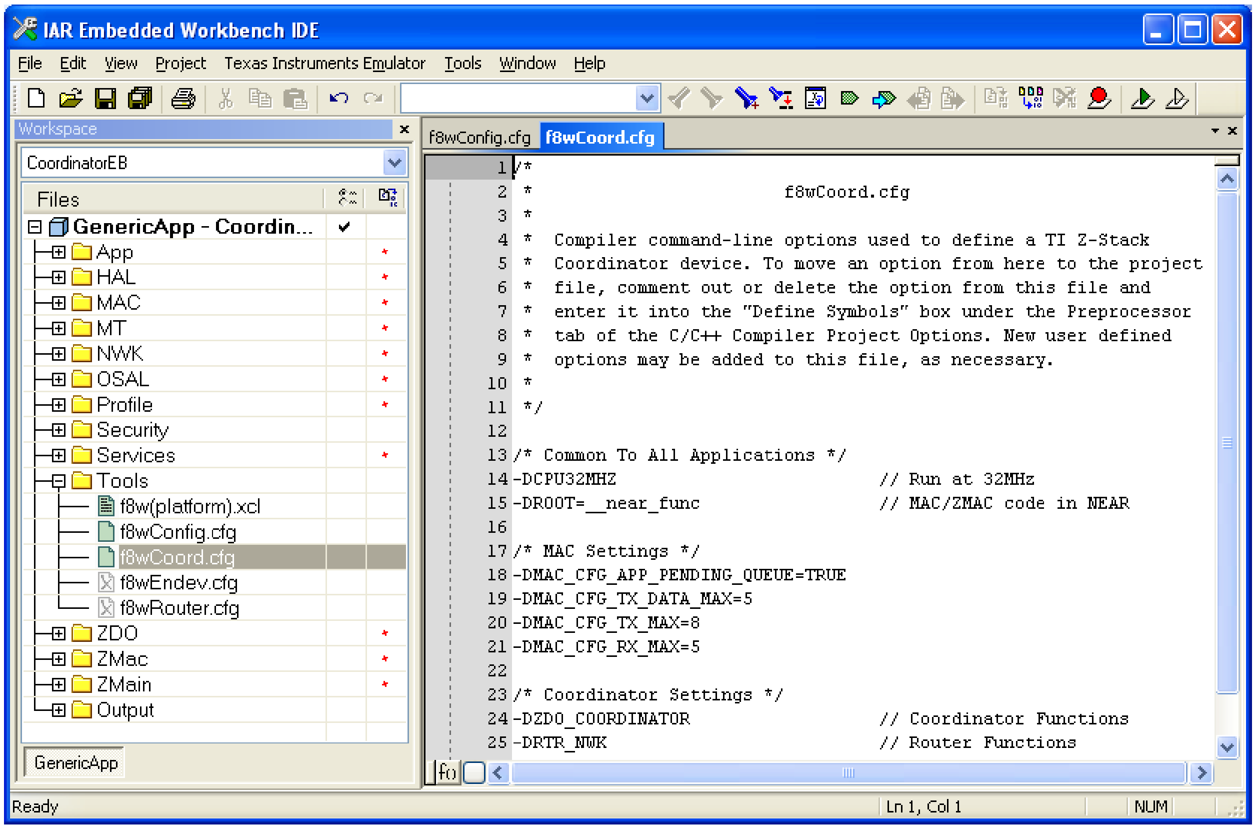
Open the project by double-clicking on the *GenericApp.eww* file, select the *CoordinatorEB* configuration from the pull-down list below **Workspace**, and then open the **Tools** folder. Several linker control files are located in the **Tools** folder. This folder contains various configuration files and executable tools used in Z-Stack projects. Generic compile options are defined in the `f8wConfig.cfg` file. This file, for example, specifies the PAN ID that will be used when a device starts up. Device specific compile options are located in the `f8wCoord.cfg`, `f8wEndev.cfg`, and `f8wRouter.cfg` files:



```
//-DMAX_CHANNELS_24GHZ       0x07FFF800
//-DDEFAULT_CHANLIST=0x04000000  // 26 - 0x1A
//-DDEFAULT_CHANLIST=0x02000000  // 25 - 0x19
//-DDEFAULT_CHANLIST=0x01000000  // 24 - 0x18
//-DDEFAULT_CHANLIST=0x00800000  // 23 - 0x17
//-DDEFAULT_CHANLIST=0x00400000  // 22 - 0x16
//-DDEFAULT_CHANLIST=0x00200000  // 21 - 0x15
//-DDEFAULT_CHANLIST=0x00100000  // 20 - 0x14
//-DDEFAULT_CHANLIST=0x00080000  // 19 - 0x13
//-DDEFAULT_CHANLIST=0x00040000  // 18 - 0x12
//-DDEFAULT_CHANLIST=0x00020000  // 17 - 0x11
//-DDEFAULT_CHANLIST=0x00010000  // 16 - 0x10
//-DDEFAULT_CHANLIST=0x00008000  // 15 - 0x0F
//-DDEFAULT_CHANLIST=0x00004000  // 14 - 0x0E
//-DDEFAULT_CHANLIST=0x00002000  // 13 - 0x0D
//-DDEFAULT_CHANLIST=0x00001000  // 12 - 0x0C
//-DDEFAULT_CHANLIST=0x00000800  // 11 - 0x0B

/* Define the default PAN ID.
 *
 * Setting this to a value other than 0xFFFF causes
 * ZDO_COORD to use this value as its PAN ID and
 * Routers and end devices to join PAN with this ID
 */
-DZDAPP_CONFIG_PAN_ID=0xFFFF

/* Minimum number of milliseconds to hold off the start of the device
 * in the network and the minimum delay between joining cycles.
```

Figure 24: Specifying the channels in f8wConfig.cfg

The GenericApp Coordinator project uses the `f8wCoord.cfg` file. As shown below, compile options that are specific to Coordinator devices and options that provide "generic" Z-Stack functions are included in this file:



```
/*
 *                          f8wCoord.cfg
 *
 *  Compiler command-line options used to define a TI Z-Stack
 *  Coordinator 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.
 *
 */

/* Common To All Applications */
-DCPU32MHZ                              // Run at 32MHz
-DROOT=__near_func                      // MAC/ZMAC code in NEAR

/* MAC Settings */
-DMAC_CFG_APP_PENDING_QUEUE=TRUE
-DMAC_CFG_TX_DATA_MAX=5
-DMAC_CFG_TX_MAX=8
-DMAC_CFG_RX_MAX=5

/* Coordinator Settings */
-DZDO_COORDINATOR                       // Coordinator Functions
-DRTR_NWK                               // Router Functions
```

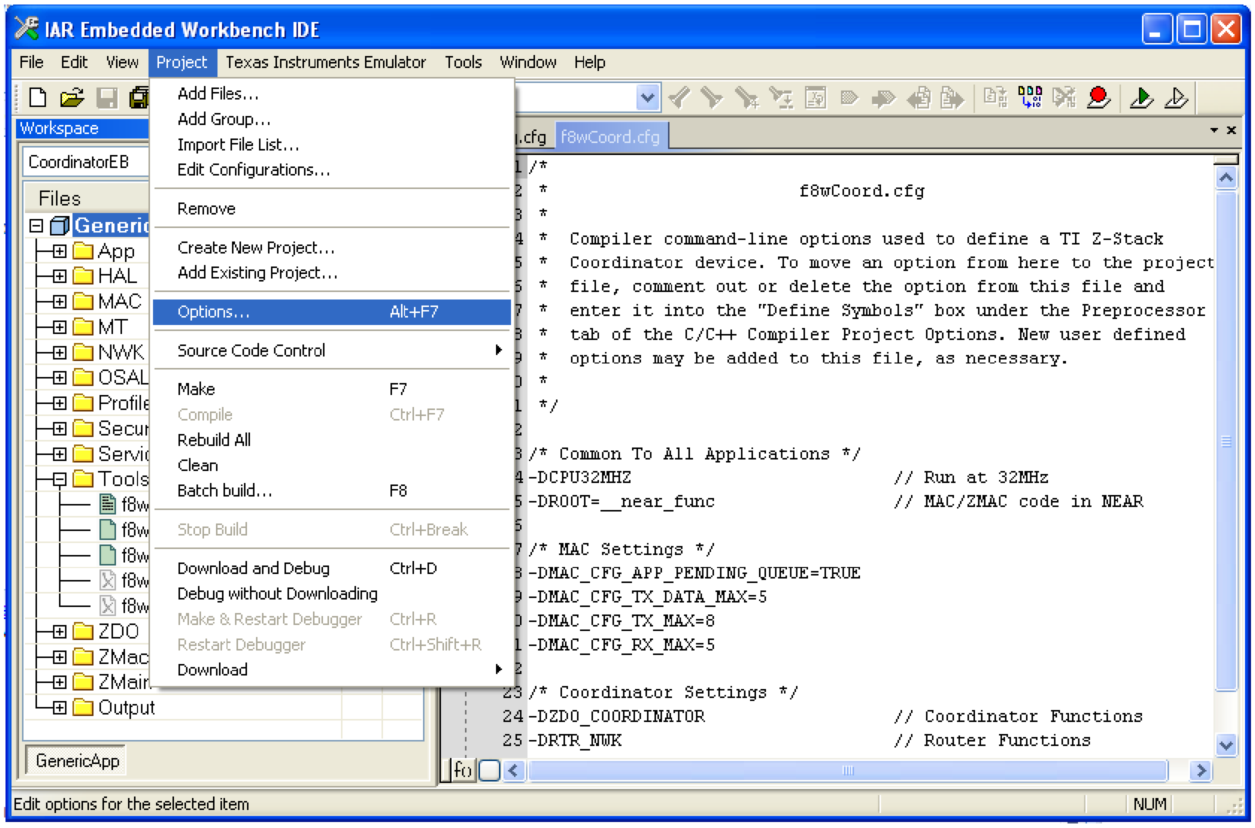
Figure 25: Coordinator settings in f8wCoord.cfg

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.

21.3.1.2 Compile Options in IAR Project Files

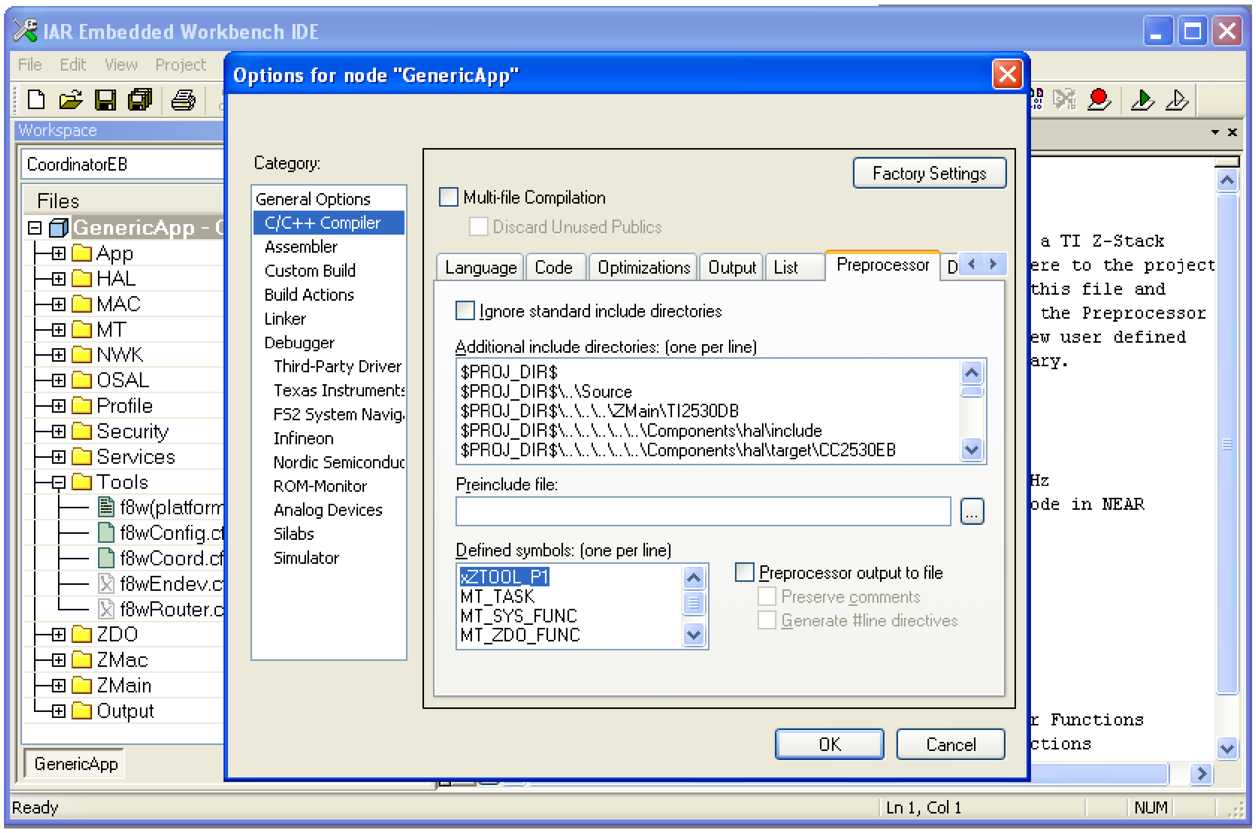
The compile options for each of the supported configurations are stored in the *GenericApp.ewp* file. To modify these compile options, first select **GenericApp – CoordinatorEB**. Then select the **Options…** item from the **Project** pull-down menu:



```
/*
 *                    f8wCoord.cfg
 *
 *  Compiler command-line options used to define a TI Z-Stack
 *  Coordinator 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.
 *
 */

/* Common To All Applications */
-DCPU32MHZ                         // Run at 32MHz
-DROOT=__near_func                 // MAC/ZMAC code in NEAR

/* MAC Settings */
-DMAC_CFG_APP_PENDING_QUEUE=TRUE
-DMAC_CFG_TX_DATA_MAX=5
-DMAC_CFG_TX_MAX=8
-DMAC_CFG_RX_MAX=5

/* Coordinator Settings */
-DZDO_COORDINATOR                  // Coordinator Functions
-DRTR_NWK                          // Router Functions
```

Figure 26: Selecting compile options

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)*:



Multi-file Compilation
Discard Unused Publics
Ignore standard include directories

Additional include directories: (one per line)

```
$PROJ_DIR$
$PROJ_DIR$\..\Source
$PROJ_DIR$\..\..\..\ZMain\TI2530DB
$PROJ_DIR$\..\..\..\..\..\Components\hal\include
$PROJ_DIR$\..\..\..\..\..\Components\hal\target\CC2530EB
```

Preinclude file:

Defined symbols: (one per line)

```
xZTOOL_P1
MT_TASK
MT_SYS_FUNC
MT_ZDO_FUNC
```

Preprocessor output to file
Preserve comments
Generate #line directives

Figure 27: Enabling or disabling options

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 the **ZTOOL_P1** option has been disabled in the example shown above. This option could have been deleted but this is not recommended since it might need to be re-enabled at a later time.

21.3.2 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. Note that changing compile options may require other changes to the project file (see 21.3.1). For example, adding the `MT_NWK_FUNC` option requires `MT_NWK.c` to be in the list of source files in the configuration of the device you are building.

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.

21.4 Supported Compile Options and Definitions

21.4.1 General Compile Options

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
BLINK_LEDS	Enable extended LED blinking functions
DEFAULT_CHANLIST	Change this list in f8wConfig.cfg
EXTENDED_JOINING_RANDOM_MASK	Mask for the random joining delay
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
NONWK	Disable NWK, APS, and ZDO functionality
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)
ZAPP_Px	Enable ZApp messages via serial port Px where x is the port (1 or 2)
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
ZTOOL_Px	Enable ZTool messages via serial port Px where x is the port (1 or 2)
OSC32K_CRYSTAL_INSTALLED	This compilation flag defines whether to use the internal 32 Khz RC OSC (if set to false) or an external 32 Khz crystal when mounted on board**.** **Important note: If this compilation flag is not defined, the SW select the 32 Khz external OSC configuration by default** (as 253x EM have an external 32 Khz populated). If your design doesn't have an external 32 Khz on board, please make sure you set **OSC32K_CRYSTAL_INSTALLED = FALSE** in your compiler project settings.

21.4.2 Non-changeable Compile Options

These compile options in the following table should not be changed or used. Not all of them are available in every platform:

CPU32MHZ	Clock rate of the CPU – 32 MHZ **(do not change)**
MACSIM	Enable MAC simulation **(do not use)**
NWK_TEST	Enable Network test functions **(do not use)**

21.4.3 Monitor-Test (MT) Compile Options

Please read the Z-Stack Monitor and Test API document before changing any of these compile options. You can enable the following APIs and function associated with the `MT_TASK` option, but you must include the `MT_TASK` option.

MT_TASK	Enable Monitor-Test task
MT_AF_FUNC	Enable Monitor-Test processing of AF commands issued from ZTool or ZTrace
MT_AF_CB_FUNC	Enable Monitor-Test processing of AF callbacks registered by ZTool or ZTrace
MT_APP_FUNC	Enable Monitor-Test processing of APP commands issued from ZTool or ZTrace
MT_DEBUG_FUNC	Enable Monitor-Test processing of DEBUG commands issued from ZTool or ZTrace
MT_MAC_FUNC	Enable Monitor-Test processing of MAC commands issued from ZTool or ZTrace
MT_NWK_FUNC	Enable Monitor-Test processing of NWK commands issued from ZTool or ZTrace
MT_NWK_CB_FUNC	Enable Monitor-Test processing of NWK callbacks registered by ZTool or ZTrace
MT_SAPI_FUNC	Enable Monitor-Test processing of SAPI commands issued from ZTool or ZTrace
MT_SAPI_CB_FUNC	Enable Monitor-Test processing of SAPI callbacks registered by ZTool or ZTrace
MT_SYS_FUNC	Enable Monitor-Test processing of SYS commands issued from ZTool or ZTrace

MT_SYS_OSAL_NV_READ_CERTIFICATE_DATA	Default define to FALSE in MT_SYS.c and only applicable if ZCL_KEY_ESTABLISH is defined. If ZCL_KEY_ESTABLISH is defined and MT_SYS_OSAL_NV_READ_CERTIFICATE_DATA is defined to TRUE, then the three NV items containing Certicom certificate data can be read via MT:
	`ZCD_NV_IMPLICIT_CERTIFICATE 0x0069`
	`ZCD_NV_DEVICE_PRIVATE_KEY 0x006A`
	`ZCD_NV_CA_PUBLIC_KEY 0x006B`
	Otherwise, the certificate data cannot be read via MT.

MT_UTIL_FUNC	Enable Monitor-Test processing of UTIL commands issued from ZTool or ZTrace
MT_ZDO_CB_FUNC	Enable Monitor-Test processing of ZDO commands issued from ZTool or ZTrace
MT_ZDO_FUNC	Enable Monitor-Test processing of ZDO commands issued from ZTool or ZTrace
MT_ZDO_MGMT	Enable Monitor-Test processing of ZDO MGMT commands from ZTool or ZTrace
MT_APP_CNF_FUNC	Enable Monitor-Test processing of BDB API interface from ZTool
MT_GP_CB_FUNC	Enable Monitor-Test processing of GP API interface from ZTool

21.4.4 ZigBee Device Object (ZDO) Compile Options

By default, the mandatory messages (as defined by the ZigBee spec and BDB) are enabled in the ZDO. To review the mandatory messages enabled by BDB refer to ZDConfig.h. 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 `MT_ZDO_FUNC` to enable all the ZDO Function options, and

MT_ZDO_FUNC and MT_ZDO_MGMT to enable all the ZDO Function plus Management options. 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.