Z-Stack Overview

Introduction

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.

Scope

This document describes concepts and settings for the Texas Instruments Z-Stack™ Release. This is a Zigbee PRO 2017 (R22) certified stack for the Zigbee and Zigbee PRO stack profiles. It also exlains the added features of the Z3.0 specification and how they can be used for compatibility with Z3.0 or legacy devices.

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

Term	Definition
BSP	Board Support Package – taken together, HAL & OSAL comprise a rudimentary op
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

Reference Documents

1. Zigbee document 05-3474-22 Zigbee PRO 2017 (R22) Specification
2. Zigbee document 07-5123-07 Zigbee Cluster Library 7 Specification

3. Zigbee document 13-0402-13 Zigbee Base Device Behavior
4. Zigbee document 14-0563-16 Zigbee Green Power specification

Zigbee

A Zigbee network is a multi-hop network of mains-powered or battery-powered devices. This means that successful communication between two devices may require intermediate devices to relay messages. Due to the cooperative nature of the network, each device is required to perform specific networking functions and configure certain parameters to specific values. The role of a device is determined by the set of networking functions it performs and is called the **logical device type**. The set of parameters that need to be configured to specific values, along with those values, is called the **stack profile**.

Logical Device Types

The three logical device types in a Zigbee network are *Coordinator*, *Router*, and *End Device*. A Zigbee network consists of a device with network 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.

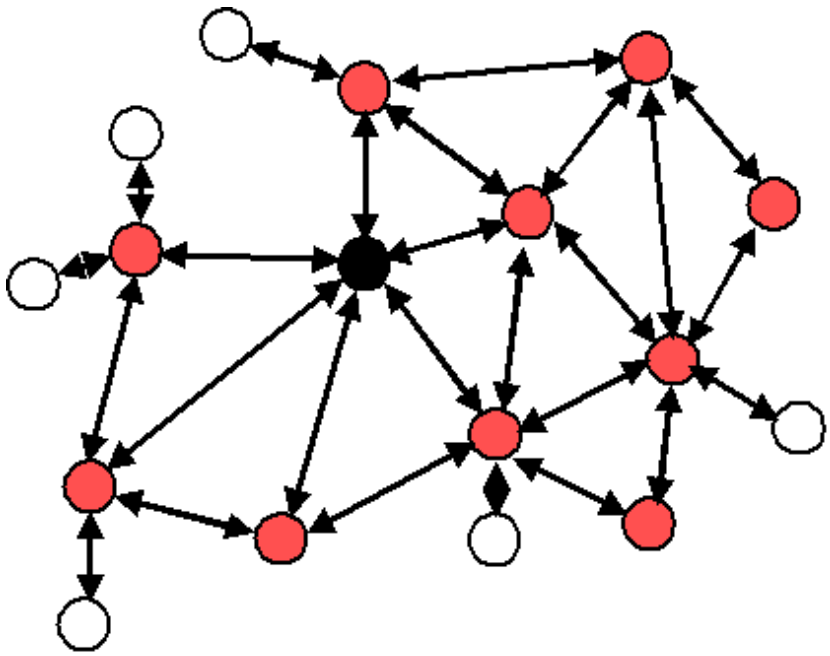

Figure 54. Example of typical Zigbee network

Example of typical Zigbee network shows a typical Zigbee network with the Zigbee *Coordinator* (black), the *Routers* (red), and the *End Devices* (white).

An application can be defined as any of these three logical devices depending on the configuration flags enabled in the project.

Logical Device	Compilation flags needed
Coordinator	`ZDO_COORDINATOR` and `RTR_NWK`
Router	`RTR_NWK`
End Device	None

Coordinator

A coordinator is a device with network formation capabilities, but without network joining capabilities. This means the device 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 for managing security of the network and 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. Note that the Coordinator must handle the network procedures related to devices joining or leaving the network, so it cannot be absent of its own network. Further details on security schema are available in Security.

Router

A router performs functions for allowing other devices to join the network, for multi-hop routing, and for assisting its child end devices with communication. 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 Security.

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

End Device

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

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

> **Note**
>
> In Z-Stack all sample application projects are provided with the needed pre-include files to build each device type according to the project.

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 profiles.

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, the legacy Zigbee stack profile has ID of 1, and the 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 the `nwk_globals.h` file. The stack profile ID of 3 is reserved for Green Power devices, and it appears in the respective frames.

Addressing

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 IEEE Registration Authority.

The 16-bit address is assigned to a device when it joins a network. Within that network, it is unique and used for identifying devices and sending data.

Network Address Assignment

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 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” frame (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 router node 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, but 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.

Addressing in Z-Stack

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

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

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

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

The address mode parameter is necessary because, in Zigbee, packets can be unicast, multicast, or broadcast. A unicast packet is sent to a single device, a multicast packet is destined to a group of devices and a broadcast packet is generally sent to all devices in the network. An indirect packet is used when the application does not explicitly know the destination of the packet. This is explained in more detail below.

Unicast

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

Indirect

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

Broadcast

This address mode is used when the application wants to send a packet to all devices in the network. The address mode is set to `zstack_AFAddrMode_BROADCAST` and the destination address can be set to one of the following broadcast addresses:

`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 `zstack_config.h`).

`NWK_BROADCAST_SHORTADDR_DEVRXON` (0xFFFD) – the message will be sent to all devices that have the receiver on when idle (RXONWHENIDLE), that is, all non-sleepy devices.

`NWK_BROADCAST_SHORTADDR_DEVZCZR` (0xFFFC) – the message is sent to all routers (including the coordinator).

Group Addressing

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

Before using this feature, groups must be defined in the network (see `Zstackapi_ApsAddGroupReq()` in the Z-Stack API).

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 set up ahead of time and defined by Zigbee Alliance.

Sample code for a device to add itself to a group with identifier `0x0001`:

```
#define GROUP_NAME "Group1"
zstack_apsAddGroup_t group;

group.endpoint = SAMPLEAPP_ENDPOINT;

/* Assign yourself to group 1 */
group.groupID = 0x0001;

/* First byte is string length */
group.n_name[0] = 6;

osal_memcpy( &(group.n_name[1]), GROUP_NAME, 6);
Zstackapi_ApsAddGroupReq(appEntity, &group);
```

Important Device Addresses

An application may want to know the address of a device (self or remote device). Use the following functions to get the addresses.

- `Zstackapi_ZdoNwkAddrReq()` – Use IEEE address to retrieve the short address.
- `Zstackapi_ZdoIeeeAddrReq()` – Use Short address to retrieve the IEEE address.

> **Note**
>
> The responses to this messages are provided by the stack with the command IDs `zstackmsg_CmdIDs_ZDO_NWK_ADDR_RSP` and `zstackmsg_CmdIDs_ZDO_IEEE_ADDR_RSP`

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 to the destination application (or multiple applications) or group.

Building a Binding Table

There are 3 ways to build a binding table:

- Zigbee Device Object Bind Request – a commissioning tool can tell the device to make a binding record.
- Zigbee Device Object End Device Bind Request – 2 devices can tell the coordinator that they would like to setup a binding table record. The coordinator will make the match up and create the binding table entries in the 2 devices.
- Finding and Binding commissioning process for initiator devices.

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.

The Commissioning Application

An application can create a bind between two remote devices by calling `Zstackapi_ZdoBindReq()` defined in `zstackapi.h` for which are needed the addresses, 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) . The remaining parameters are of the remote application device that the bind will use to send frames. Calling `Zstackapi_ZdoUnbindReq()` 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. The ZDO code on the coordinator will parse this and notify the application with the message `zstackmsg_CmdIDs_ZDO_BIND_RSP` or `zstackmsg_CmdIDs_ZDO_UNBIND_RSP`.

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`.

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 is 16 seconds (see `APS_DEFAULT_MAXBINDING_TIME` in `zglobals.h`), but can be changed if added to `zstack_config.h` or as a compile flag.

Coordinator end device binding is a toggle process. 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.

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 performs 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 a 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.

End Device Binding process is already built in the coordinator stack and does not require application interaction.

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 that these pairs of devices can communicate more effectively. Refer to section Finding and Binding for further details on this commissioning method.

Configuring Source Binding

To enable source binding in your device include the `REFLECTOR` compile flag in `zstack_config.h`. Also in `zstack_config.h`, 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 directly impacts 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.

Routing

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 hidden from the application layer. The application simply sends data destined to any device down to the stack which is then responsible for finding a route. In other words, 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 detail in section Many-to-One Routing Protocol.

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 `zstack_config.h`. 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.

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: - Route discovery and selection - Route maintenance - Route expiry These are explained in more detail below.

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.

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.

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.

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 `zstack_config.h`. Set `ROUTE_EXPIRY_TIME` to expiry time in seconds. Set to 0 in order to turn off route expiry feature.

Table Storage

The routing functions require the routers to maintain some tables.

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 `zstack_config.h`. Set `MAX_RTG_ENTRIES` to the number of entries in the (default is 40). See the section on Route Maintenance for route expiration details.

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 `zstack_config.h`.

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.

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.

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.

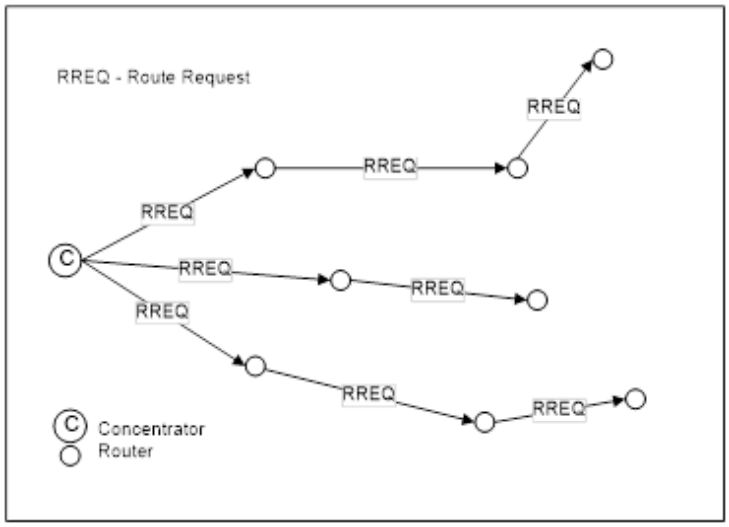

Figure 55. Many-to-One route discovery illustration

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 documentation for detailed usage about this API.

```
zstack_ZStatusValues Zstackapi_DevNwkRouteReq(
      appServiceTaskId, zstack_devNwkRouteReq_t *pReq)
```

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 memo
0x03	Many-to-One route discovery with no route cache (the concentrator has memory const

When the option field has value 0x01 or 0x03, the DstAddress field will be overwritten with the Many-to-One destination address 0xFFFC. Therefore, in a Many-to-One request, it is irrelevant what value the application sets the DstAddress to.

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 is provided in the Route record command (source routing) illustration. *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.

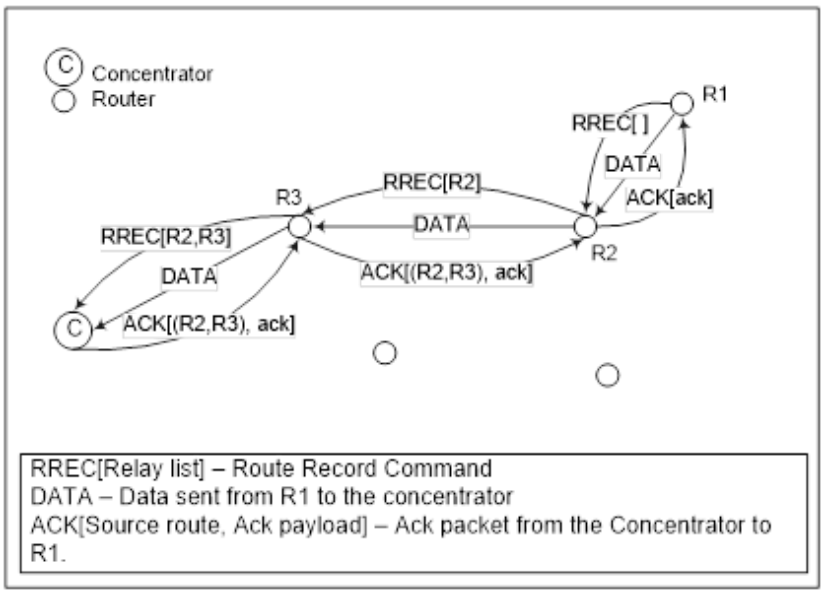

Figure 56. Route record command (source routing) illustration

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.

A concentrator with no memory constraints 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 a route record command once. However, for a 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 requests to the concentrator, they should also send a route record along with it.

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 `zd_app.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.

Routing Settings Quick Reference

Setting Routing Table Size	Set `MAX_RTG_ENTRIES` Note: the value must be greater than 4
Setting Route Expiry Time	Set `ROUTE_EXPIRY_TIME` to expiry time in secc
Setting Route Discovery Table Size	Set `MAX_RREQ_ENTRIES` to the maximum num
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 `nwk_globals.h`)
Setting Default Concentrator Broadcast Radius	Set `CONCENTRATOR_RADIUS` (See `zglobals.h`)

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_t 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.

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, as it is managed by the stack.

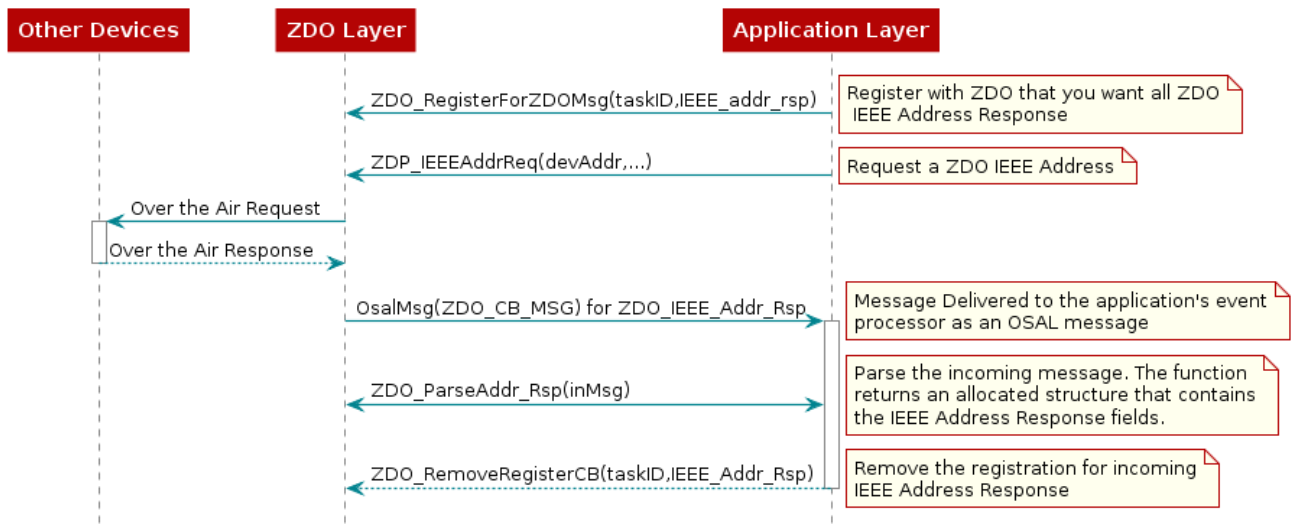

Figure 57. ZDO IEEE Address Request and Response

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.

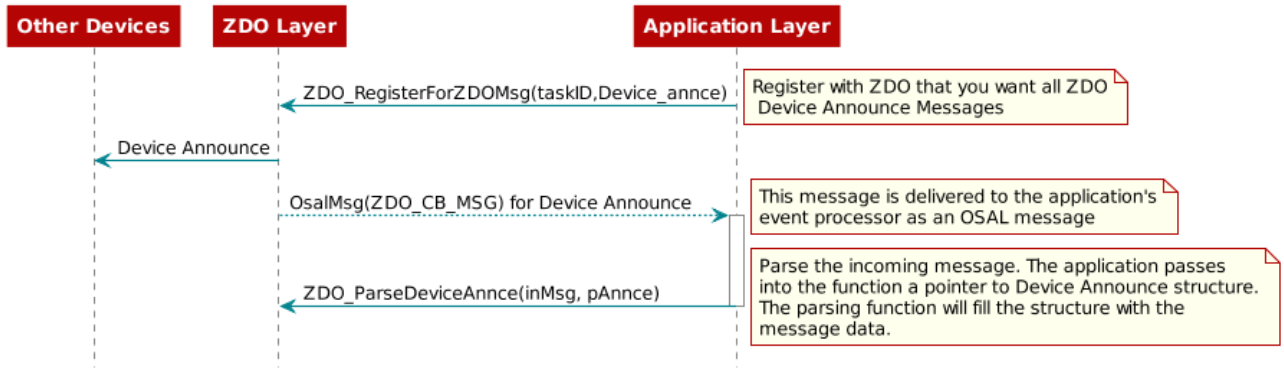

Figure 58. ZDO Device Announce Delivered to an Application

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 calling `Zstackapi_sysConfigWriteReq()`. In the `zstack_sysConfigWriteReq_t` argument, set `has_pollFailureRetries` to `TRUE` and `pollFailureRetries` to the number of failures (the higher the number, the less sensitive and the longer it will take to rejoin).

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 Parent Lost). The application is responsible for managing the rejoining of the device by using the BDB API `stackapi_bdbZedAttemptRecoverNwkReq()`, 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.

End-to-End Acknowledgements

For non-broadcast messages, there are basically 2 types of message retries: 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 to 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. When sending a message with `AF_DataRequest`, APS ACK for that message can be enabled by setting the `AF_ACK_REQUEST` bit in the `options` argument. 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 `zstack_config.h`. `APSC_MAX_FRAME_RETRIES` is the maximum number of times the APS layer will retry sending a message if it doesn't receive an APS ACK. `APSC_ACK_WAIT_DURATION_POLLED` is the amount of time between retries.

The following examples demonstrate how end-to-end acknowledgements can be configured and handled within a `zcl_samplesw` project.

- Add a zcl options struct

```
static zclOptionRec_t zclOptions[] =
{
  {
    ZCL_CLUSTER_ID_GEN_ON_OFF,
    ( AF_ACK_REQUEST ),
  },
};
```

- Define a global variable to save APS counter, `uint8 savedAPSCount;`
- Place `zcl_registerClusterOptionList(SAMPLESW_ENDPOINT,1,zclOptions);` inside of `zclSampleSw_Init()` to register the cluster options.
- Save the APS counter (`savedAPSCount = APS_Counter;`) before sending the toggle command (`zclGeneral_SendOnOff_CmdToggle( SAMPLESW_ENDPOINT, &zclSampleSw_DstAddr, FALSE, Rsp.zclFrameCounter );`)
- Add `zstackmsg_CmdIDs_AF_DATA_CONFIRM_IND` processing to `zclSampleSw_processZStackMsgs()`

```
case zstackmsg_CmdIDs_AF_DATA_CONFIRM_IND:
{
  zstackmsg_afDataConfirmInd_t *pInd =
  (zstackmsg_afDataConfirmInd_t *) pMsg;
  if(savedAPSCount == pInd->req.transID)
  {
    // PERFORM ACTION HERE
  }
}
break;
```

All APS layer commands will generate an AF data confirm. The difference between using the `APS_ACK_REQUEST` flag is that a `zstackmsg_CmdIDs_AF_DATA_CONFIRM_IND` is received only when the APS ACK indication arrives. Otherwise, with the APS ACK flag disabled, `zstackmsg_CmdIDs_AF_DATA_CONFIRM_IND` occurs whenever the stack sends out an APS layer frame successfully.

Miscellaneous

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. If the process is not successful (the network was not created or no network to join was found), then 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 as `DEFAULT_CHANLIST` (in `zstack_config.h`), and the secondary channel mask is defined as all the other channels (i.e. `DEFAULT_CHANLIST ^ 0x07FFF800`). Section Commissioning provides more details on the commissioning methods.

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 that a coordinator or router will create. The `ZDAPP_CONFIG_PAN_ID` parameter in `zstack_config.h` 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 in Network Steering Procedure for a Node not on a Network. It allows filtering of the discovered networks. After the scan (using either primary or secondary channel masks) is complete, the application receives a list of network descriptors of the networks found during the scan (signified by a BDB Z-stack message

`zstackmsg_CmdIDs_BDB_FILTER_NWK_DESCRIPTOR_IND`). The application may skip attempting to join specific networks by freeing the corresponding network descriptors using `Zstackapi_bdbNwkDescFreeReq()`.

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

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 bytes (can be changed in `zstack_config.h` – see `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 (see `afDataReqMTU()` `AF.h`) that allows the user to query the stack for the maximum payload size, or the maximum transport unit (MTU).

```
typedef struct
{
    uint8_t kvp;
    APSDE_DataReqMTU_t aps;
} afDataReqMTU_t;

uint8_t 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. This field should be set to `FALSE`. The aps field is reserved for future use.

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 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, it 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.

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 `zd_config.c` and `zd_config.h`. These 2 files also contain the Node, Power Descriptors, and default User Descriptor. Make sure to change these descriptors to define your device.

Non-Volatile Memory Items

Global Configuration Non-Volatile Memory

Global device configuration items are stored in `zglobals.c`. This includes items such as PAN ID, key information, network settings, etc. The default values for most of these items are specified in `zstack_config.h`. These items are loaded to RAM at startup for quick access 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).

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 is only intended for use 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 a "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.

Upon reset, if the device is not meant to return to its factory new state, then it will use this information to restore itself in the network.

Upon initializing, the BDB layer will check the attribute if this device was commissioned to a network (`bdbNodeIsOnANetwork`). 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.

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.

Reading and writing to NV is done using the NV functions contained within `zstack_user0Cfg.nvFps`. The sample applications have access to these functions via the global `static NVINTF_nvFuncts_t *pfnZdlNV = NULL;`

The NV area of flash is used for storing persistent data for the application. Z-Stack provides two implementations of NV. One uses one page of internal flash, while the other uses two. By default example applications use one page NV. For more information on one page NV please refer to `nvocop.c` which describes the implementation details. Also `nvoctp.c` describes the implementation details of the two page NV and lists the maximum values of custom NV IDs available to the applications as such:

```
// Maximum ID parameters - must be coordinated with header compression,
// Increasing these limits requires modification of the readHdr() function
#define NVOCTP_MAXSYSID   0x003F  //  6 bits
#define NVOCTP_MAXITEMID  0x03FF  // 10 bits
#define NVOCTP_MAXSUBID   0x03FF  // 10 bits
#define NVOCTP_MAXLEN     0x03FF  // 10 bits
```

The last page in flash is the CCA page, depending on whether one page or two page NV is used one or two pages before the last page (CCFG) are defined as the NV area. The example projects use the NV driver with the API defined in `nvintf.h`. The NV driver is set up in `taskFxn` of `main.c`:

```
#ifdef NV_RESTORE
    /* Setup the NV driver */
    NVOCTP_loadApiPtrs(&zstack_user0Cfg.nvFps);

    if(zstack_user0Cfg.nvFps.initNV)
    {
        zstack_user0Cfg.nvFps.initNV( NULL);
    }
#endif
```

Then the applications use the function pointers in `zstack_user0Cfg` to call the NV functions defined in `nvintf.h`:

```
//! Structure of NV API function pointers
typedef struct nvintf_nvfuncts_t
{
    //! Initialization function
    NVINTF_initNV initNV;
    //! Compact NV function
    NVINTF_compactNV compactNV;
    //! Create item function
    NVINTF_createItem createItem;
    //! Delete NV item function
    NVINTF_deleteItem deleteItem;
    //! Read item function
    NVINTF_readItem readItem;
    //! Write item function
    NVINTF_writeItem writeItem;
    //! Write existing item function
    NVINTF_writeItemEx writeItemEx;
    //! Get item length function
    NVINTF_getItemLen getItemLen;
} NVINTF_nvFuncts_t;
```

The following is an example of NV memory registration from `zcl_sampledoorlock.c`:

```
void sampleApp_task(NVINTF_nvFuncts_t *pfnNV)
{
  // Save and register the function pointers to the NV drivers
  pfnZdlNV = pfnNV;
  zclport_registerNV(pfnZdlNV, ZCL_PORT_SCENE_TABLE_NV_ID);

  // Initialize application
  zclSampleDoorLock_initialization();

  // No return from task process
  zclSampleDoorLock_process_loop();
}
```

The following is an example of a NV write from `zclSampleDoorLock_Init` of `zcl_sampledoorlock.c`:

```
// Initialize NVM for storing PIN information
if(pfnZdlNV)
{
    NVINTF_itemID_t nvId;
    uint32_t nvErr = NVINTF_NOTFOUND;

    // Fill in the NV ID header
    nvId.systemID = NVINTF_SYSID_APP;
    nvId.itemID = (uint16_t)DLSAPP_NV_DOORLOCK_PIN;
    nvId.subID = (uint16_t)0;

    // Read the PIN from NV
    if(pfnZdlNV->readItem)
    {
        nvErr = pfnZdlNV->readItem(nvId, 0, DLSAPP_NV_DOORLOCK_PIN_LEN,
                                   aiDoorLockMasterPINCode);
    }

    // If the PIN doesn't exist in NV, create it
    if((nvErr == NVINTF_NOTFOUND) && pfnZdlNV->createItem)
    {
        pfnZdlNV->createItem(nvId, DLSAPP_NV_DOORLOCK_PIN_LEN,
                             aiDoorLockMasterPINCode);
    }
}
```

The following is an example of a read from `zclSampleDoorLock_UiActionChangePin` of `zcl_sampledoorlock.c`:

```
if(pfnZdlNV && pfnZdlNV->writeItem)
{
    NVINTF_itemID_t nvId;

    nvId.systemID = NVINTF_SYSID_APP;
    nvId.itemID = (uint16_t)DLSAPP_NV_DOORLOCK_PIN;
    nvId.subID = (uint16_t)0;

    pfnZdlNV->writeItemEx(nvId, 0, DLSAPP_NV_DOORLOCK_PIN_LEN,
                          aiDoorLockMasterPINCode);
}
```

The NV system is a collection of NV items. Each item is unique and have the following pieces to it (defined in `nvintf.h`):

```
/**
*  NV Item Identification structure
*/

typedef struct nvintf_itemid_t
{
    //! NV System ID - identifies system (ZStack, BLE, App, OAD...)
    uint8_t systemID;
    //! NV Item ID
    uint16_t itemID;
    //! NV Item sub ID
    uint16_t subID;
} NVINTF_itemID_t;
```

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 a device doesn't received a Link Status message from a neighbor within (`_NIB.nwkRouterAgeLimit` * `_NIB.nwkLinkStatusPeriod`), the device will age the neighbor out and assume that this neighbor is missing or that it's an asynchronous link and not use it.

Multicast Messages

This feature is a Zigbee PRO only feature (must have `ZIGBEEPRO_PROFILE` as the `STACK_PROFILE_ID`). 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, which includes a non-member radius field. The receiving device will determine if it is part of that group. If it isn't part of the group, then it will decrement the non-member radius and rebroadcast. If it is part of the group, then it will first set the non-member radius equal to the group radius, and then rebroadcast the message. If the non-member 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`) for all Group messages. If this field is `FALSE`, then the APS Group message is sent as a normal broadcast network message.

`zgApsNonMemberRadius` is the value of the group radius and the initial value of the non-member radius. The application should use this variable 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.

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 `ZIGBEEPRO_PROFILE` projects 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 the following:

The header file `aps_frag.h`:

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

An entry for `APSF_ProcessEvent()` in `zstackTasksArr[]`:

```
const pTaskEventHandlerFn zstackTasksArr[] =
{
#ifdef NPI
  MT_ProcessEvent,
#endif
  ZMacEventLoop,
  nwk_event_loop,
#if !defined (DISABLE_GREENPOWER_BASIC_PROXY) && (ZG_BUILD_RTR_TYPE)
  gp_event_loop,
#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
//Added to include TouchLink functionality
#if defined ( INTER_PAN ) || defined ( BDB_TL_INITIATOR ) || defined ( BDB_TL_TARGET )
  StubAPS_ProcessEvent,
#endif
// Added to include TouchLink initiator functionality
#if defined ( BDB_TL_INITIATOR )
  touchLinkInitiator_event_loop,
#endif
// Added to include TouchLink target functionality
#if defined ( BDB_TL_TARGET )
  touchLinkTarget_event_loop,
#endif
    bdb_event_loop,
    ZStackTaskProcessEvent
};
```

A call to `APSF_Init()` in And `stackServiceFxnsInit()`:

```
static void stackServiceFxnsInit( void )
{
    uint8_t tmpServiceId, idx = 0;

    pTasksEvents = MAP_osal_mem_alloc( sizeof( uint32_t ) * zstackTasksCnt);

#ifdef NPI
    tmpServiceId = OsalPort_registerTask(stackTaskHndl, stackSemHandle, &mtServiceEvents);
    MT_TaskInit( tmpServiceId );
    pTasksEvents[idx] = &mtServiceEvents;
    idx++;
#endif

    tmpServiceId = OsalPort_registerTask(stackTaskHndl, stackSemHandle, &macServiceEvents);
    MAP_macMainSetTaskId( tmpServiceId );
    pTasksEvents[idx] = &macServiceEvents;
    idx++;

    tmpServiceId = OsalPort_registerTask(stackTaskHndl, stackSemHandle, &nwkServiceEvents);
    nwk_init( tmpServiceId );
    pTasksEvents[idx] = &nwkServiceEvents;
    idx++;

#if !defined (DISABLE_GREENPOWER_BASIC_PROXY) && (ZG_BUILD_RTR_TYPE)
    tmpServiceId = OsalPort_registerTask(stackTaskHndl, stackSemHandle, &gpServiceEvents);
    gp_Init( tmpServiceId );
    pTasksEvents[idx] = &gpServiceEvents;
    idx++;
#endif

    tmpServiceId = OsalPort_registerTask(stackTaskHndl, stackSemHandle, &apsServiceEvents);
    APS_Init( tmpServiceId );
    pTasksEvents[idx] = &apsServiceEvents;
    idx++;

#if defined ( ZIGBEE_FRAGMENTATION )
    tmpServiceId = OsalPort_registerTask(stackTaskHndl, stackSemHandle, &apsfServiceEvents);
    APSF_Init( tmpServiceId );
    pTasksEvents[idx] = &apsfServiceEvents;
    idx++;
#endif

    tmpServiceId = OsalPort_registerTask(stackTaskHndl, stackSemHandle, &zdAppServiceEvents);
    ZDApp_Init( tmpServiceId );
    pTasksEvents[idx] = &zdAppServiceEvents;
    idx++;

#if defined ( ZIGBEE_FREQ_AGILITY ) || defined ( ZIGBEE_PANID_CONFLICT )
    tmpServiceId = OsalPort_registerTask(stackTaskHndl, stackSemHandle, &zdpNwkMgrServiceEvents);
    ZDNwkMgr_Init( tmpServiceId );
    pTasksEvents[idx] = &zdpNwkMgrServiceEvents;
    idx++;
#endif

    // Added to include TouchLink functionality
#if defined ( INTER_PAN )
    tmpServiceId = OsalPort_registerTask(stackTaskHndl, stackSemHandle, &stubApsServiceEvents);
    StubAPS_Init( tmpServiceId );
    pTasksEvents[idx] = &stubApsServiceEvents;
    idx++;
```

```
#endif

      // Added to include TouchLink initiator functionality
#if defined( BDB_TL_INITIATOR )
    tmpServiceId = OsalPort_registerTask(stackTaskHndl, stackSemHandle,
&touchLinkInitiatorServiceEvents);
    touchLinkInitiator_Init( tmpServiceId );
    pTasksEvents[idx] = &touchLinkInitiatorServiceEvents;
    idx++;
#endif

      // Added to include TouchLink target functionality
#if defined ( BDB_TL_TARGET )
    tmpServiceId = OsalPort_registerTask(stackTaskHndl, stackSemHandle,
&touchLinkTargetServiceEvents);
    touchLinkTarget_Init( tmpServiceId );
    pTasksEvents[idx] = &touchLinkTargetServiceEvents;
    idx++;
#endif

    tmpServiceId = OsalPort_registerTask(stackTaskHndl, stackSemHandle, &bdbServiceEvents);
    bdb_Init( tmpServiceId );
    pTasksEvents[idx] = &bdbServiceEvents;
    idx++;

    stackServiceId = OsalPort_registerTask(stackTaskHndl, stackSemHandle, &zstackServiceEvents);
    ZStackTaskInit( stackServiceId );
    pTasksEvents[idx] = &zstackServiceEvents;

#if defined(FLASH_ROM_BUILD)
  // initialize the Common ROM
  CommonROM_Init();

  // initialize the TIMAC ROM
  ROM_Init();
#endif /* FLASH_ROM_BUILD */

  return;
}
```

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. 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. For example, 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.

The application can read and set these values 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 (see `ZDConfig_UpdateNodeDescriptor()` in `zd_config.c`). These fields are initialized with `MAX_TRANSFER_SIZE` (defined in `zd_config.h`). These values are not used in the APS layer as maximums; they are information only.

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 `zgloba`
Application/Profile maximum buffer size	`MAX_TRANSFER_SIZE` (defined in `zd_config.h`)

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.

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 Commissioning for rejoining the network.

Child Management

R21 (revision 21 of the Zigbee PRO specification, AKA Zigbee 2015), has introduced a child management feature. This feature allows mobility in end devices and saves space within parent devices' association tables. When an end device joins/rejoins, it will send an EndDeviceTimeout nwk command, which contains a timeout period. (If their parent does not receive a keep-alive message from the end device within the timeout period, it can remove the end device from its association table and send that end device a leave request). 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`. Then this device can rejoin the network and find a new parent (which could be the same router or another one).

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 it should send leave requests to. To do this, parent devices must queue leave requests in the MAC layer. The number of devices that can be kept track of at the same time is defined by `MAX_NOT_MYCHILD_DEVICES` (in `nwk_globals.h`). These devices will be tracked for a period of time defined by `NWK_END_DEVICE_LEAVE_TIMEOUT` (in `zglobals.h`).

Configuring Child Management for Child Devices

The timeout that the child device will indicate to its parent device is defined by `END_DEV_TIMEOUT_VALUE` (in `zglobals.h`). Its suggested value should 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.

Parent Annce

The child management functionality includes the usage of Parent Annce ZDO messages. A parent broadcasts this message only when forming a network or when reset, after 10 seconds plus a random jitter of up to 10 seconds. The message contains the 64-bit IEEE

addresses of all end devices in the parent's association table. If another parent device receives this message, it will check if any of the reported children is also listed in its own association table. If there are any matches, then this parent device will respond to the originator of the message with a list of all matches. The originator will then remove those children from its association table. 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 device 'A' 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 'A' removes child 'c' from its table.

Security

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 schema 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 default value for Z-Stack is 180 seconds as determined by `BDBC_MIN_COMMISSIONING_TIME` in `bdb_interface.h`. 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 device's beacon requests.

Zigbee offers the following security features:

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

Configuration

Network layer security is mandatory in Z3.0 and cannot be disabled in Z-Stack

The default key for network layer encryption (`defaultKey` defined in `nwk_globals.c`) is distributed to each joining device over-the-air as they join the network. This is chosen via the `zgPreConfigKeys` option in `zglobals.c`, where it is set to `FALSE` such that 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 `zstack_config.h`. 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.

Centralized Security Network

This network type is formed by coordinator devices, in which the coordinator assumes the role of Trust Center (TC). In this type of network only the TC can deliver the network key to joining devices 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 Trust Center Policies. 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 device's 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.

Trust Center Policies

zgAllowRemoteTCPolicyChange

If `zgAllowRemoteTCPolicyChange` of `zglobals.c` is set to `TRUE` (default), 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.

bdbJoinUsesInstallCodeKey

If `BDB_DEFAULT_JOIN_USES_INSTALL_CODE_KEY` from `bdb_interface.h` is set to `TRUE`, then the network key will be delivered only to those joining devices that do have an install code associated. If `BDB_DEFAULT_JOIN_USES_INSTALL_CODE_KEY` is set to `FALSE` (default), joining devices may use install codes. The usage of install codes is described in Install Code Derived Trust Center Link Key.

bdbTrustCenterRequireKeyExchange

If `BDB_DEFAULT_TC_REQUIRE_KEY_EXCHANGE` in `bdb_interface.h` is set to `TRUE`, then 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` (default), 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 Unsecure Join to a Centralized Network.

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.

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 `zd_sec_mgr.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(s). Once the Trust Center calls `ZDSecMgrSwitchNwkKey()` with the dstAddr of the device or devices, all destination devices will use their alternate key.

The application may use the `Zstackapi_sec` functions to request ZDO Security Manager features. Here is a list of functions available to the application:

- `Zstackapi_secNwkKeyGetReq()`
- `Zstackapi_secNwkKeySetReq()`
- `Zstackapi_secNwkKeyUpdateReq()`
- `Zstackapi_secNwkKeySwitchReq()`
- `Zstackapi_secApsLinkKeyGetReq()`
- `Zstackapi_secApsLinkKeySetReq()`
- `Zstackapi_secApsLinkKeyRemoveReq()`
- `Zstackapi_secApsRemoveReq()`

In the R21 revision of the Zigbee specification, the network frame counter is mandated to be persistent across factory new resets. However, it can be reset to 0 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 then reset the frame counter to 0.

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 (detailed in Distributed Security Global Link Key). 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 that 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.

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. An install code derived Trust Center link key (when `BDB_DEFAULT_JOIN_USES_INSTALL_CODE_KEY` is `TRUE`).
3. The distributed security global link key (Used by Z-Stack automatically).
4. The touchlink preconfigured link key (if touchlink enabled).

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 is defined by the `DEFAULT_TC_LINK_KEY` macro from `nwk_globals.h` and 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 0x

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, 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 opened.
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.

This is accomplished by setting `BDB_DEFAULT_JOIN_USES_INSTALL_CODE_KEY` to `TRUE` in `bdb_interface.h` and using the `Zstackapi_bdbAddInstallCodeReq()` API. For further details on how to generate the install codes, see the Base Device Behavior Specification [3]. 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.

Distributed Security Global Link Key

When a device joins a distributed security network (no TC), the parent router device sends the network key after encrypting it in the APS layer using the Distributed Global link key, defined as `DISTRIBUTED_GLOBAL_LINK_KEY` found in `nwk_globals.h`. 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

Touchlink Preconfigured Link Key

This key is used for development of a device that will join a network using the Touchlink commissioning procedure.

Touchlink preconfigured link key (0:15)	=	0xc0 0xc1 0xc2 0xc3 0xc4 0xc5 0xc6 0xc7 0xc8 0

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. The procedure specifies how the joining device discovers networks across one set of channels, and how it can fall back to discover additional networks in the remaining channels, refer to Network Steering Procedure for a Node not on a Network. Once the device has selected a suitable network, the joining device determines if it has joined a Centralized or Distributed security network. These networks use different keys to encrypt the APS Transport command. The specific secure procedures to join these types of secure networks will be explained in the following subsections.

Unsecure Join to a Centralized Network

Once the joining device receives the transport key, it will proceed to check the source address of that transport key command. In this case the 64-bit IEEE address will be different from all 00's or 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 59..

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 Trust Center Link Key or an Install Code Derived Trust Center Link Key.
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 [1]).
   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 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 status from *Default* or *Provisional* (if an 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 hashes the key associated to this device and compares against the hashed key received. If they are the same, it will send an APS Confirm Key command with status *Success*, after which the TCLK exchange procedure is finished for the joining device.

If any steps between 1 and 4 fail, the joining device will reattempt steps 1 to 4 up to `BDBC_REC_SAME_NETWORK_RETRY_ATTEMPS` (`bdb_interface.h`) times with the same network. If there are no successes within these attempts, the joining device will retry with the next suitable network in the network descriptor list. Similarly, if any steps between 5 and 8 fail, the joining device will reattempt those steps up to `BDB_DEFAULT_TC_LINK_KEY_EXCHANGE_ATTEMPS_MAX` (`bdb.h`) times. If there are no successes, then 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 of these, as detailed in BDB Notifications.

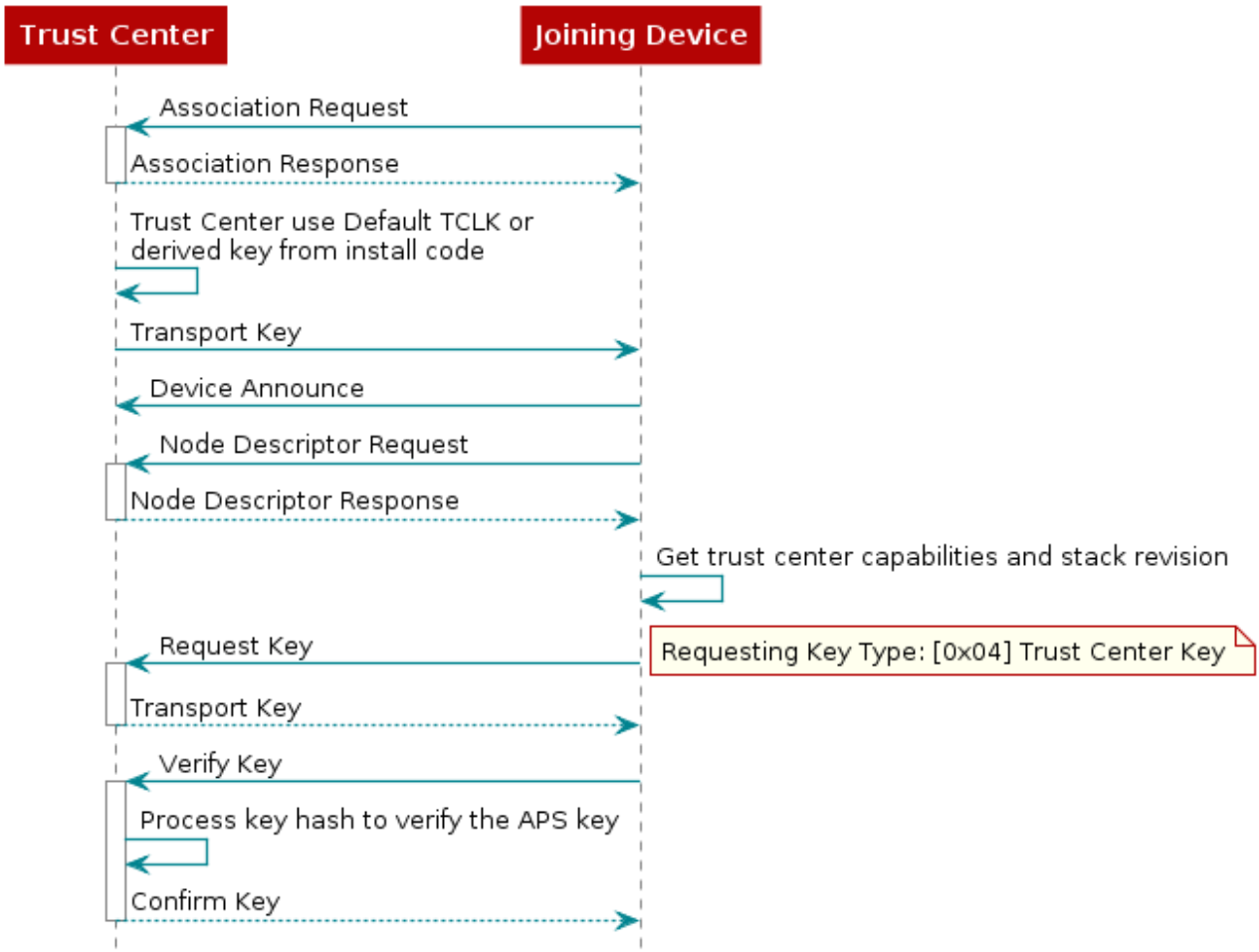

Figure 59. Joining Directly to Trust Center

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 it about the new device. Afterwards, the parent device only relays the frames between the joining device and the TC as illustrated in Figure 60..

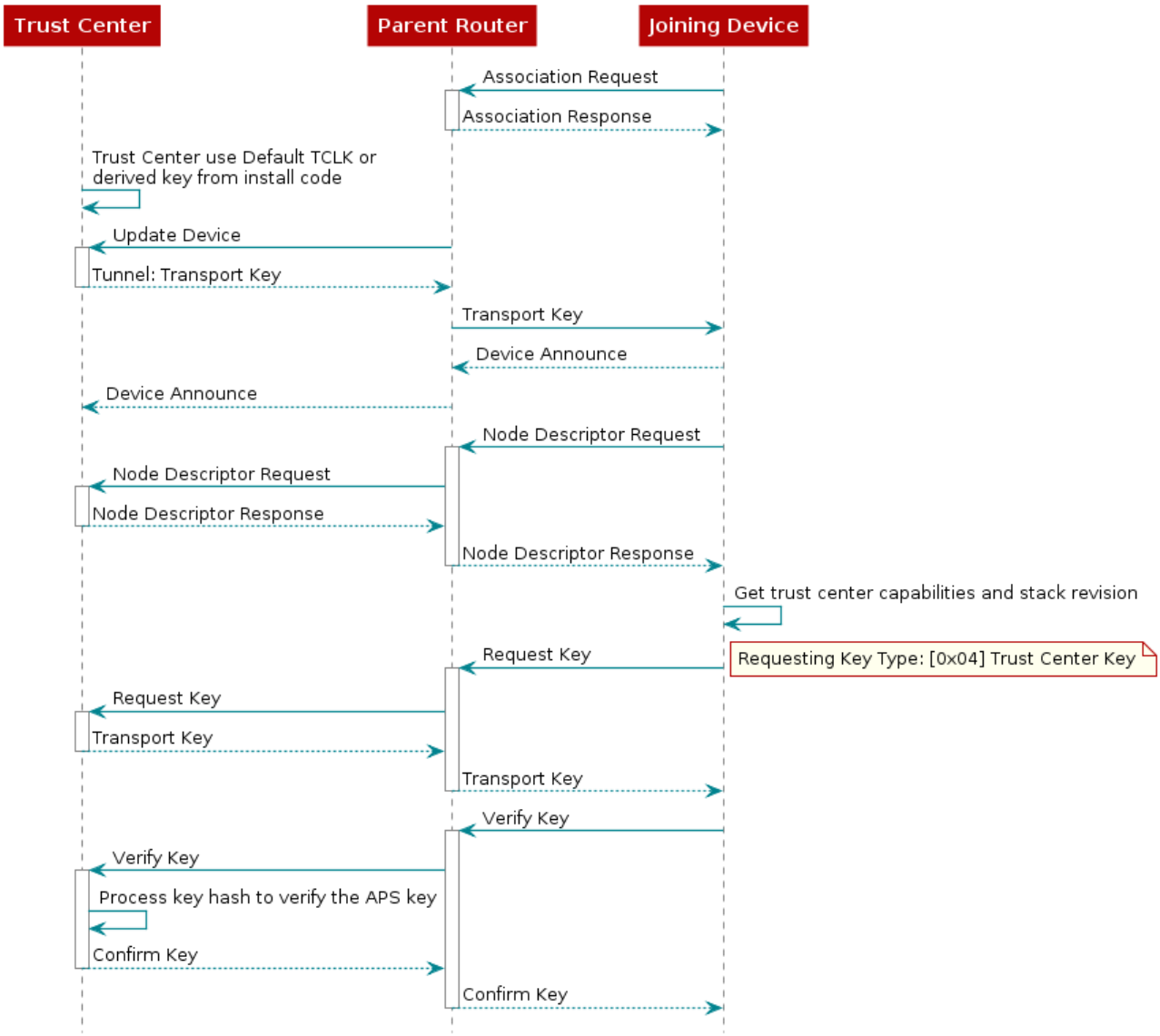

Figure 60. Joining When Parent is not the Trust Center

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 all FF's, indicating that this is a distributed network. Also, there are no additional procedures to perform updates of keys, since there is no TC that can handle this. The joining process into a Z3.0 Distributed network is illustrated in Figure 61..

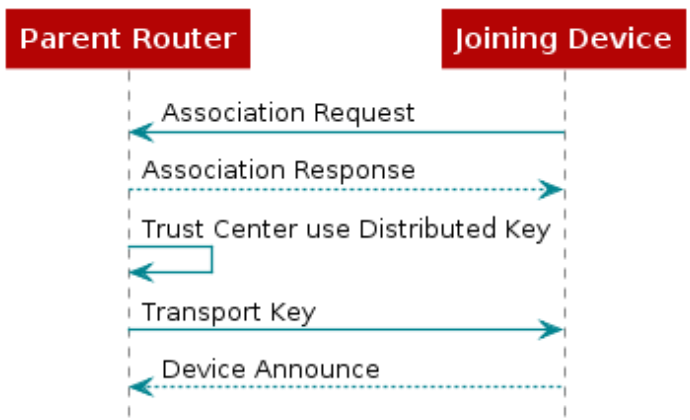

Figure 61. Distributed Security Joining

The joining device will attempt up to `BDBC_REC_SAME_NETWORK_RETRY_ATTEMPS` to join this network. If it cannot be authenticated (receives the network key), then it will try the next network in the network descriptor list.

Z-Stack Security Considerations

For Trust Center (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 TCLK entry is defined by the structure `APSME_TCLKDevEntry_t` (in `aps_mede.h`). They are stored in NV, up to a maximum defined by `ZDSECMGR_TC_DEVICE_MAX`, (in `zd_sec_mgr.h`). A TCLK entry is created for each joining device that the TC sends the network key to. This limits the number of devices in the network to the number of TCLK entries that the TC has. A TCLK entry is also used when an Install Code is introduced to the TC for a joining device, but the Install Code key is saved in a separate table of NV whose size is controlled by `ZDSECMGR_TC_DEVICE_IC_MAX` (in `zd_sec_mgr.h`). When the TCLK exchange is complete for a joining device, the TC frees the corresponding Install Code key entry from NV, but continues using the TCLK entry. However, since the TCLK entries are used to keep track of the APS Key, which 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 legacy devices. For this reason the TC will erase the TCLK entry. Then it will either kick that device out of the network or leave it in the network (depending on the configuration of `BDB_DEFAULT_TC_REQUIRE_KEY_EXCHANGE` (bdbTrustCenterRequireKeyExchange). This optimization allows a Z3.0 TC device to form a network of up to `ZDSECMGR_TC_DEVICE_MAX` Z3.0 devices and as many legacy devices as possible (limited by other parameters or topology configurations).

For Joining Devices

When a factory new device receives an APS Transport Key command, it must decrypt the contents of the command to determine which type of network it's joining (centralized or distributed). The device first assumes a centralized network, thus using its Install Code (if loaded through BDB API) or the Global Default Centralized Key to decrypt. If the decryption fails, Z-Stack will automatically try decrypting with the Global Default Distributed Key.

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 NV reads/writes 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.

A joining device without a user interface to configure its joining mechanism can be configured to try all the preconfigured keys (Install Code, Global Default Centralized Key, and Global Default Distributed Key) upon joining, by setting `ZDSECMGR_TC_ATTEMPT_DEFAULT_KEY` to `TRUE`. However, if the device is intended only to join networks which only use Install Codes, then this policy must be set to `FALSE` (default from `zd_sec_mgr.h`).

Joining devices may skip the TCLK exchange procedure by setting `requestNewTrustCenterLinkKey` inside `zd_sec_mgr.h` to `FALSE` (non-default). This allows Z3.0 devices to deploy a large custom network without requiring big tables of TCLK entries in Coordinator devices. However, this should not be used if interoperability with certified Z3.0 devices is intended.

Touchlink Joining

Touchlink commissioning is a distributed security joining procedure 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 touchlink target. If a target responds and is selected, it will be asked to form a new network for the initiator or join the initiator's network.

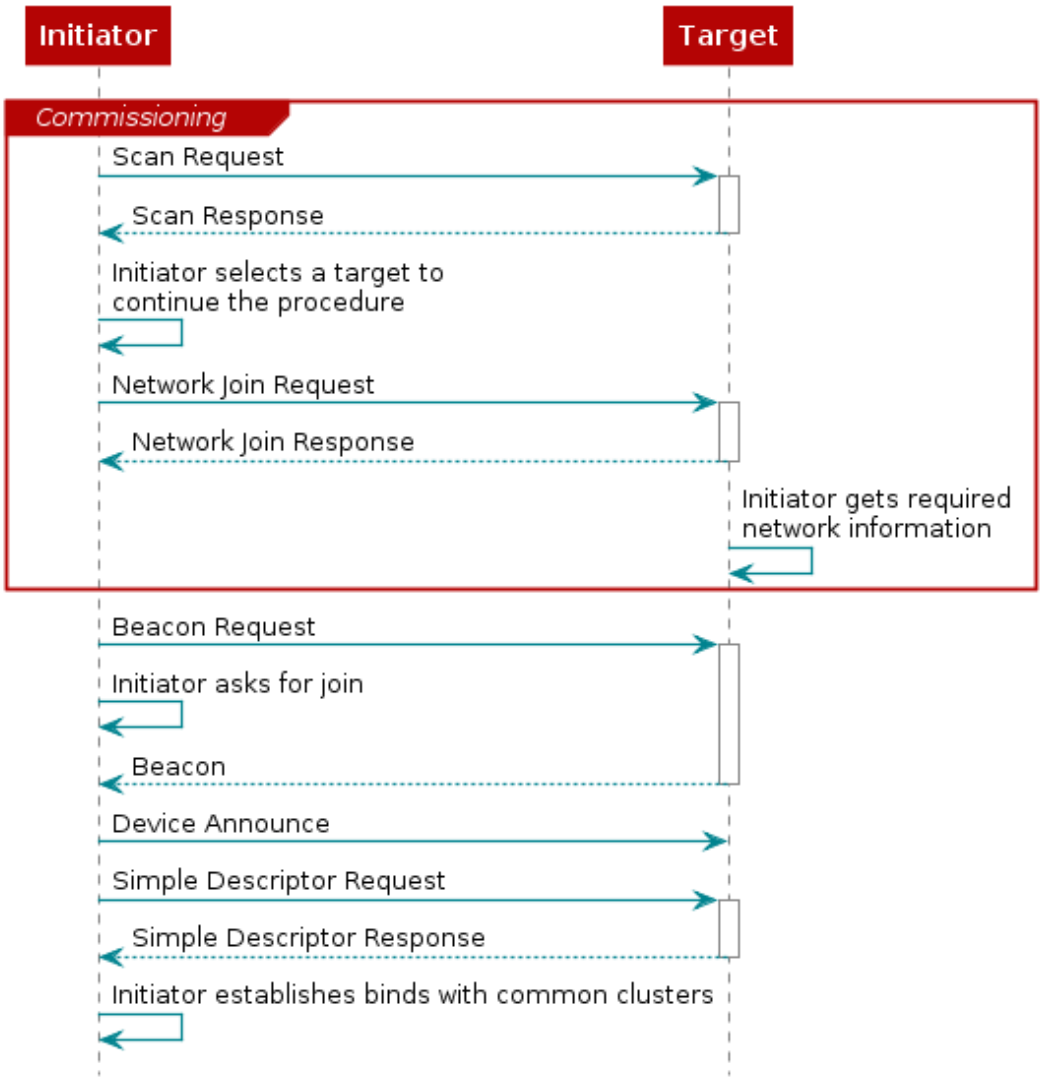

Figure 62. Asking to join with Touchlink commissioning

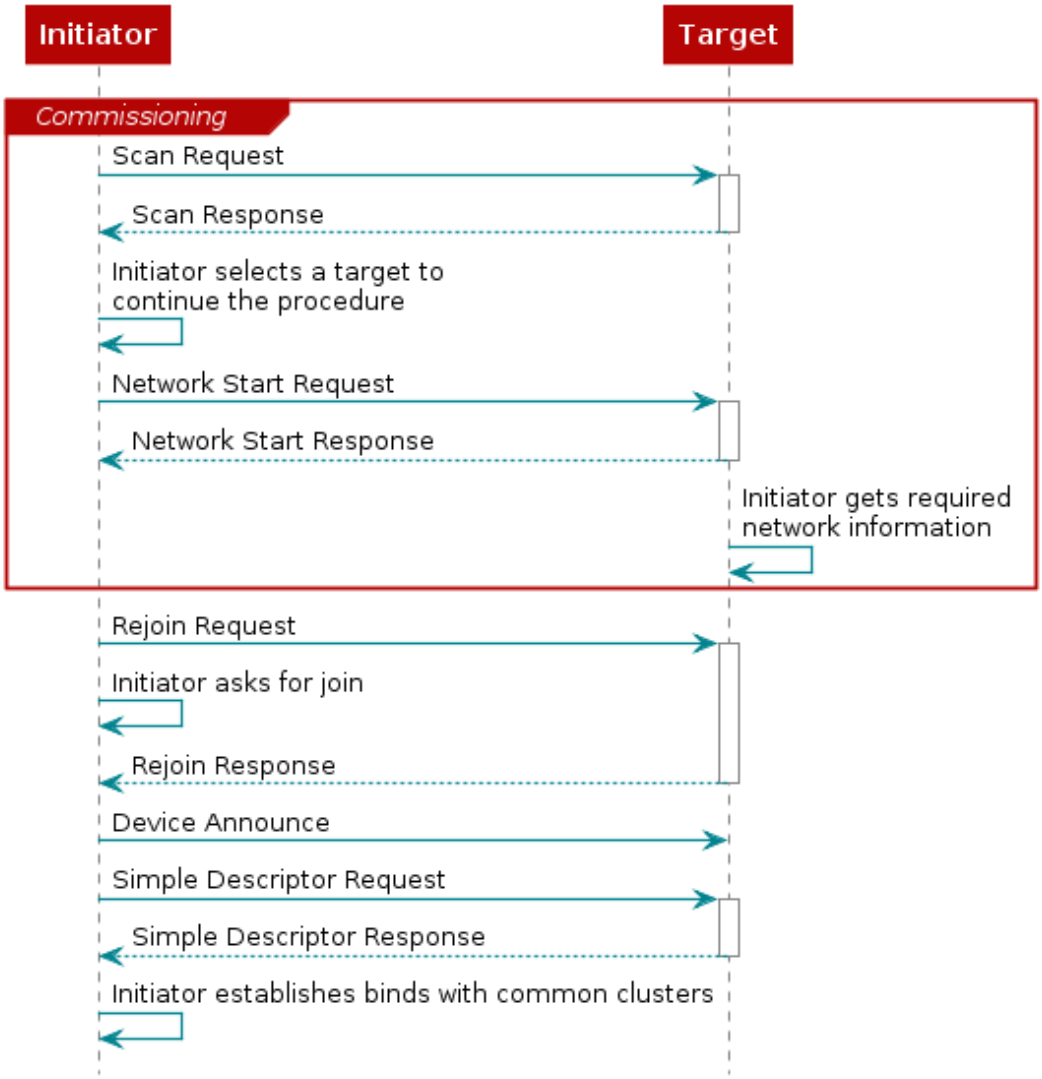

Figure 63. Asking to start network with Touchlink commissioning

Backwards Interoperability

There is a known interoperability issue that arises 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 TC only allows APS command messages with APS encryption, but Zigbee Routers running older versions of Z-Stack send APS command messages with NWK encryption only. To overcome that issue, there is a configuration control item, `zgApsAllowR19Sec` in `zglobals.c`, that the application can set to allow R19 devices to join the network (`FALSE` by default). The corresponding NV item is `ZCD_NV_APS_ALLOW_R19_SECURITY` (in `zcomdef.h`).

Quick Reference

Setting preconfigured Network key	Set `defaultKey` = {KEY} (in `nwk_globals.c`)
Enabling/disabling joining permissions	Call `Zstackapi_ZdoMgmtPermitJoinReq()`

Specific device validation during joining	Modify `ZDSecMgrDeviceValidate` (in `zd_sec_mgr.c`)
Network key updates	Call `Zstackapi_secNwkKeyUpdateReq()` and `Zstackapi_secNwk`
Use Global Trust Center Link Key	Set `zgApsLinkKeyType` = `ZG_GLOBAL_LINK_KEY` (in `zglobals.c`
Use Unique Trust Center Link Keys	Set `zgApsLinkKeyType` = `ZG_UNIQUE_LINK_KEY` (in `zglobals.c`

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. `zcl_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.

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 7 Specification [2].

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 Application Non-Volatile Memory.

Adding an Attribute Example

To add an additional attribute to a project, refer to the attributes information within the ZCL 7 Specification [2]. 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 that an application uses are defined within `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 this 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. When modifying the attribute list, keep in mind that the order of attributes in the attribute list is important. For correct processing of discovery commands, attributes of a cluster must be listed in ascending order. In other words, attributes of the same cluster must be listed one after the other, from lower attribute ID to higher.

The cluster ID can be found in 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 7 Specification [2].

By including the attribute within this list, devices are able to interact with the attributes on other devices. Within the `zcl_sampledoorlock.h` file, define the external variable using proper coding conventions:

```
extern uint8_t 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 ZCL 7 Specification.

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 to the Workspace.

Once enabled, the cluster's callbacks can be registered within the application (refer to Cluster Callbacks Example).

Cluster Architecture

All clusters follow the same architecture.

The cluster library within Z-Stack 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.

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. Listing 17. 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.

Listing 17. Cluster Callbacks Example¶

```
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
};
```

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 7 Specification [2]. 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: all reportable attributes of the same cluster share one Minimum Reporting Interval and one Maximum Reporting Interval. The shared Minimum Reporting Interval is equal to the lowest of the minimum reporting intervals of the cluster's reportable attributes. Similarly, the Maximum Reporting Interval is equal to the lowest of the maximum reporting intervals. Refer to section 2.5.11.2.5 of the ZCL 7 Specification [2] 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 with some default values. The Attribute Reporting module automatically starts (or stops) the reporting of the attributes in a cluster of an endpoint when the endpoint's bind is added (or removed).

In the BDB API (in `bdb_interface.h`) there is a method called `bdb_RepAddAttrCfgRecordDefaultToList` that adds default Attribute Reporting Configuration record values for each reportable attribute of a cluster. This API is internal to the stack, but can be accessed by the application through `Zstackapi_bdbRepAddAttrCfgRecordDefaultToListReq()`. This method must be called before the device starts the BDB Commissioning. If the application does not add default values for a given Attribute Reporting Configuration record, then global defaults values will be assigned. 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 finds the application's reportable attributes (from the attribute list) and constructs the necessary Attribute Reporting Configuration records. 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 reconsolidates the cluster's Reporting Interval values with the records contained in those messages. Calls to the `Zstackapi_bdbRepAddAttrCfgRecordDefaultToListReq` method after the BDB Commissioning has started will have no effect on the current Attribute Reporting Configuration records.

In order for the Attribute Reporting module to manage the sending of Report Attributes commands when a attribute changes value, the application must inform the module when any reportable attribute has a new value. This notification must be made by calling the `Zstackapi_bdbRepChangedAttrValueReq()` method of the Z-Stack 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.

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 BDB folder. The API interface is located in `zstackapi.h` with the prefix `Zstackapi_bdb`. The default configuration of BDB functionality is found in `bdb_interface.h`. BDB functionality may be modified at run time through the API.

The BDB interface provides an API to trigger one or more commissioning procedures defined as follows:

```
zstack_bdbStartCommissioningReq_t zstack_bdbStartCommissioningReq;
zstack_bdbStartCommissioningReq.commissioning_mode = commissioningMode;
Zstackapi_bdbStartCommissioningReq( zstack_bdbStartCommissioningReq );
```

where commissioningMode 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 above (TL as initiator first, then Nwk steering, then Formation, and lastly Finding and Binding). The priority of the tasks are checked when the current task is finished. The tasks can be appended at any time (e. g. in response to a commissioning notification). For example, suppose Nwk steering and Formation are requested. Nwk steering will start running. If TL as initiator is requested before Nwk steering is finished, then TL as initiator will start after Nwk steering is finished but before Formation starts.

There are other commissioning modes that the BDB machine state handles: **BDB_COMMISSIONING_MODE_INITIALIZATION** and **BDB_COMMISSIONING_MODE_PARENT_LOST**. These states should not be directly used by the application.

BDB Notifications

The application will receive BDB notifications from the stack automatically and will have the ID `zstackmsg_CmdIDs_BDB_NOTIFICATION`. The application can trigger another commissioning method upon receiving a certain notification. For example, a router device may start network steering to search for a suitable network and count the number of times this process fails. If this process fails 'x' times in a row, then it may decide to change the channel mask to search networks in other channels or to form its own network. The full API is described in Z-Stack API.

The notifications are called when certain tasks start or finish. Different logical devices may handle and interpret the notifications differently.

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 notifications (commissioning modes and statuses) are defined in `bdb_interface.h`. The same information is documented in the following table.

Commissioning mode (**BDB_COMMISSIONING_mode**)	Status reported (**BDB_COMMISSIONIN**
INITIALIZATION	**NETWORK_RESTORED**
NWK_STEERING (for Router and End Devices)	**IN_PROGRESS**
	NO_NETWORK
	TCLK_EX_FAILURE
	SUCCESS
NWK_STEERING (for Coordinators)	**NO_NETWORK**
	SUCCESS

FORMATION	IN_PROGRESS
	SUCCESS
	FORMATION_FAILURE
FINDING_BINDING	FB_TARGET_IN_PROGRESS
	FB_INITITATOR_IN_PROGRESS
	FB_NO_IDENTIFY_QUERY_RESPONSE
	FB_BINDING_TABLE_FULL
	FAILURE
TOUCHLINK	TL_TARGET_FAILURE
	TL_NOT_AA_CAPABLE
	TL_NO_SCAN_RESPONSE
	TL_NOT_PERMITTED
PARENT_LOST (Only for End Devices)	NO_NETWORK
	NETWORK_RESTORED

Commissioning Status Reported by the Different Commissioning Modes

Initialization Procedure

The BDB interface will perform an initialization when `Zstackapi_bdbStartCommissioningReq()` is called after a power cycle. Any commissioning mode mask may be used, and the power cycle is detected by the global RAM variable `bdb_initialization`. If the attribute `bdbNodeIsOnANetwork` is `TRUE`, the initialization procedure retrieves the network parameters from NV. Coordinator and router devices will rejoin the network and resume operations as if they never left. Upon rejoining, they will send and process parent annce messages (see Parent Annce). End devices will restore the network parameters and try to perform a rejoin on any parent available in the same network only one time. This procedure is illustrated in Figure 64. and Figure 65..

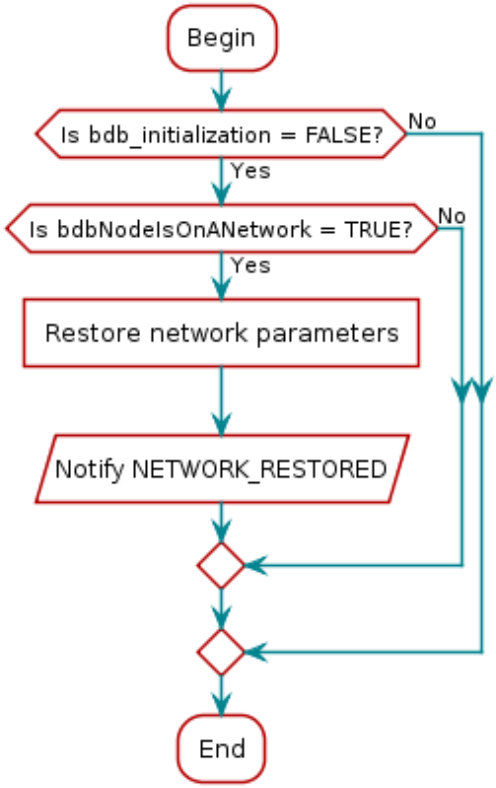

Figure 64. Initialization Procedure for a Router/Coordinator

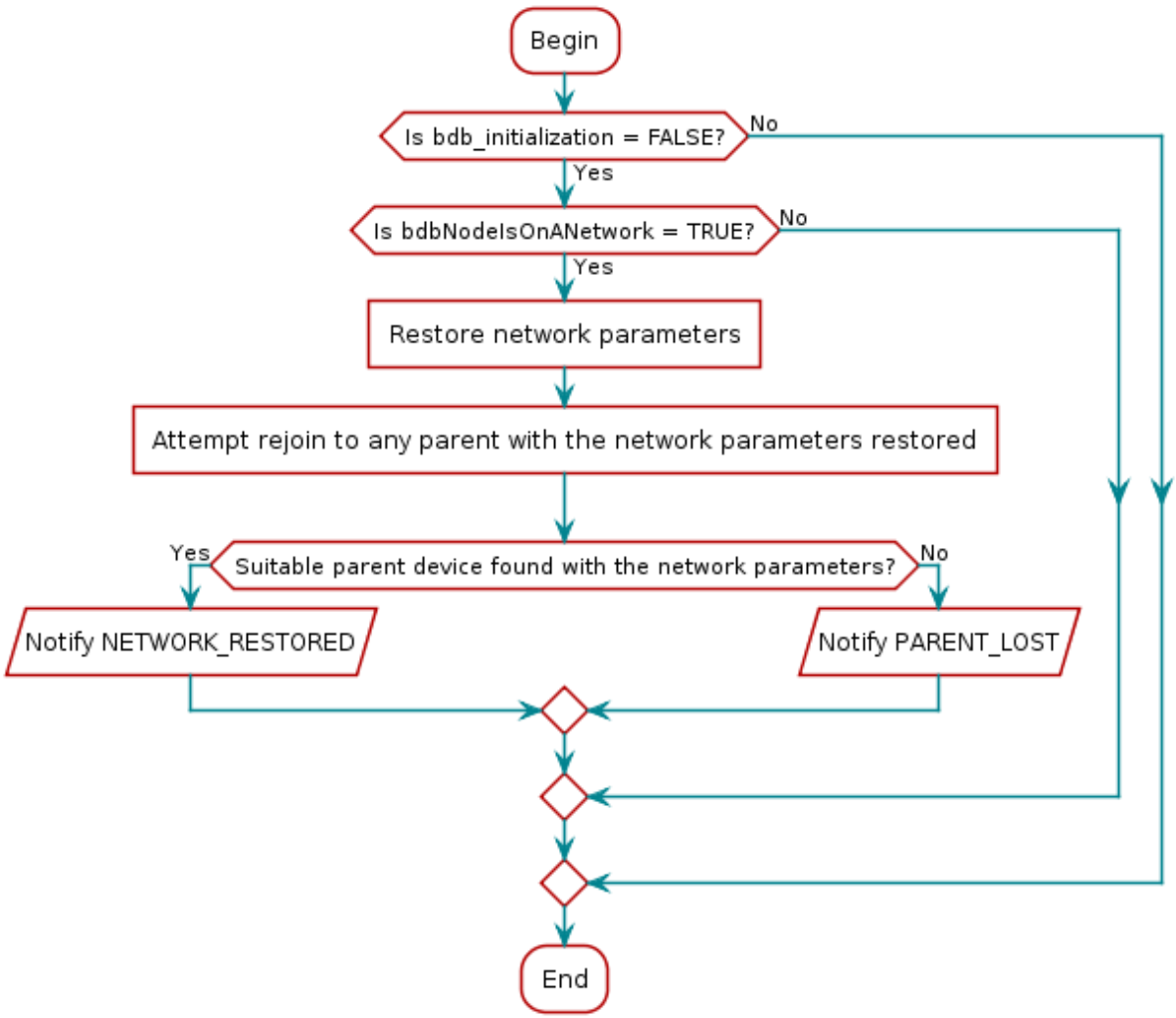

Figure 65. Initialization Procedure for a End Device

Note that if the initialization process fails for an end device it will notify the application of a `PARENT_LOST` status. Refer to Parent Lost on how to restore the network or Reset Procedures to reset the device to factory new.

Parent Lost

If an end device loses contact with its parent device or is reset while joined to a network, the BDB module will send the application a `BDB_COMMISSIONING_PARENT_LOST` notification, after which the end device cannot perform any other commissioning method. The device must either restore its network or reset to factory new and be commissioned again. To restore the network, the device must call `Zstackapi_bdbZedAttemptRecoverNwkReq()`, which performs a single active scan in the network for a suitable parent (same Extended PAN ID and child device capacity). This means that the device sends a single beacon request. If no suitable parent device is found, then another `BDB_COMMISSIONING_PARENT_LOST` notification is sent to the application. The application is responsible for attempting to restore the network. The interval between these attempts should increase to reduce power consumption. If Finding and Binding was in progress when the device lost its parent, it will keep running and resume its operation for the remaining time after the device restores its operation.

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 the application of `BDB_COMMISSIONING_SUCCESS` .

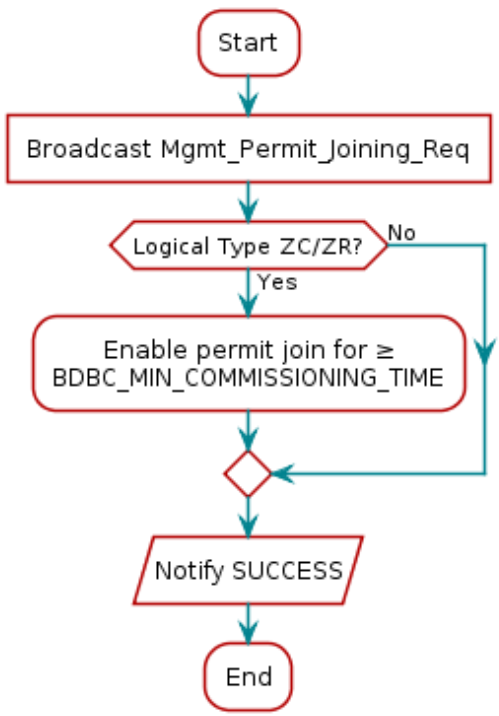

Figure 66. Network Steering Procedure for a Node On a Network

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 67. 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 will receive a zstackmsg_CmdIDs_BDB_FILTER_NWK_DESCRIPTOR_IND message, which contains a list of network descriptors of the networks found during the scan. It can use Zstackapi_bdbNwkDescFreeReq() to remove network descriptors of networks that it will not attempt to join.
   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), then 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 Security. For Centralized networks it will also perform the TCLK exchange.
3. If the joining procedure is completed, then the joining device will broadcast a permit joining request to refresh the joining timeout for other devices trying to join simultaneously. The network manager can close the network for joining by sending a permit join request with timeout = 0.

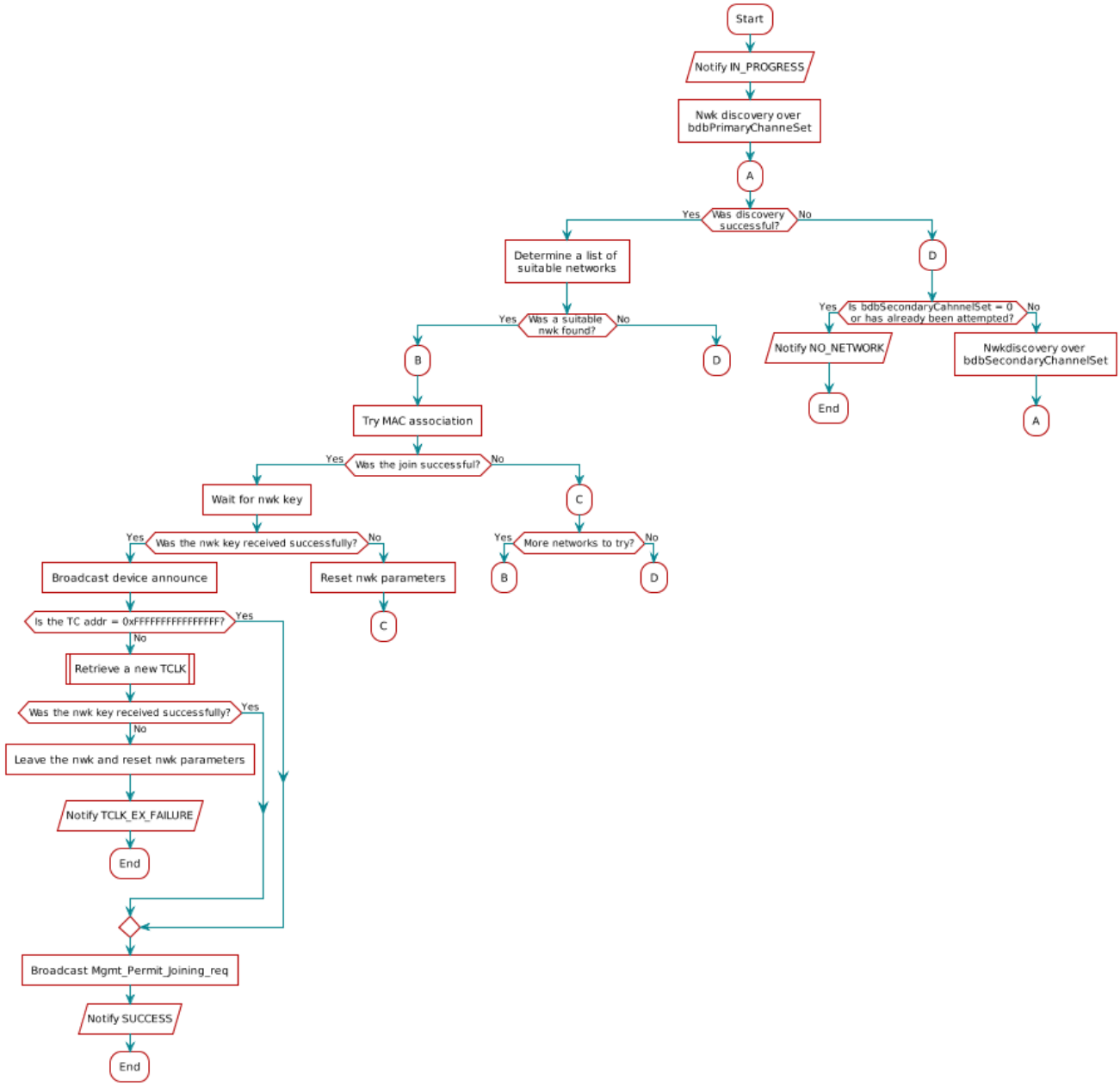

Figure 67. Network Steering Procedure for a Node Not on a Network

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. After a successful network formation, the application can open the network for joining with the network steering procedure.

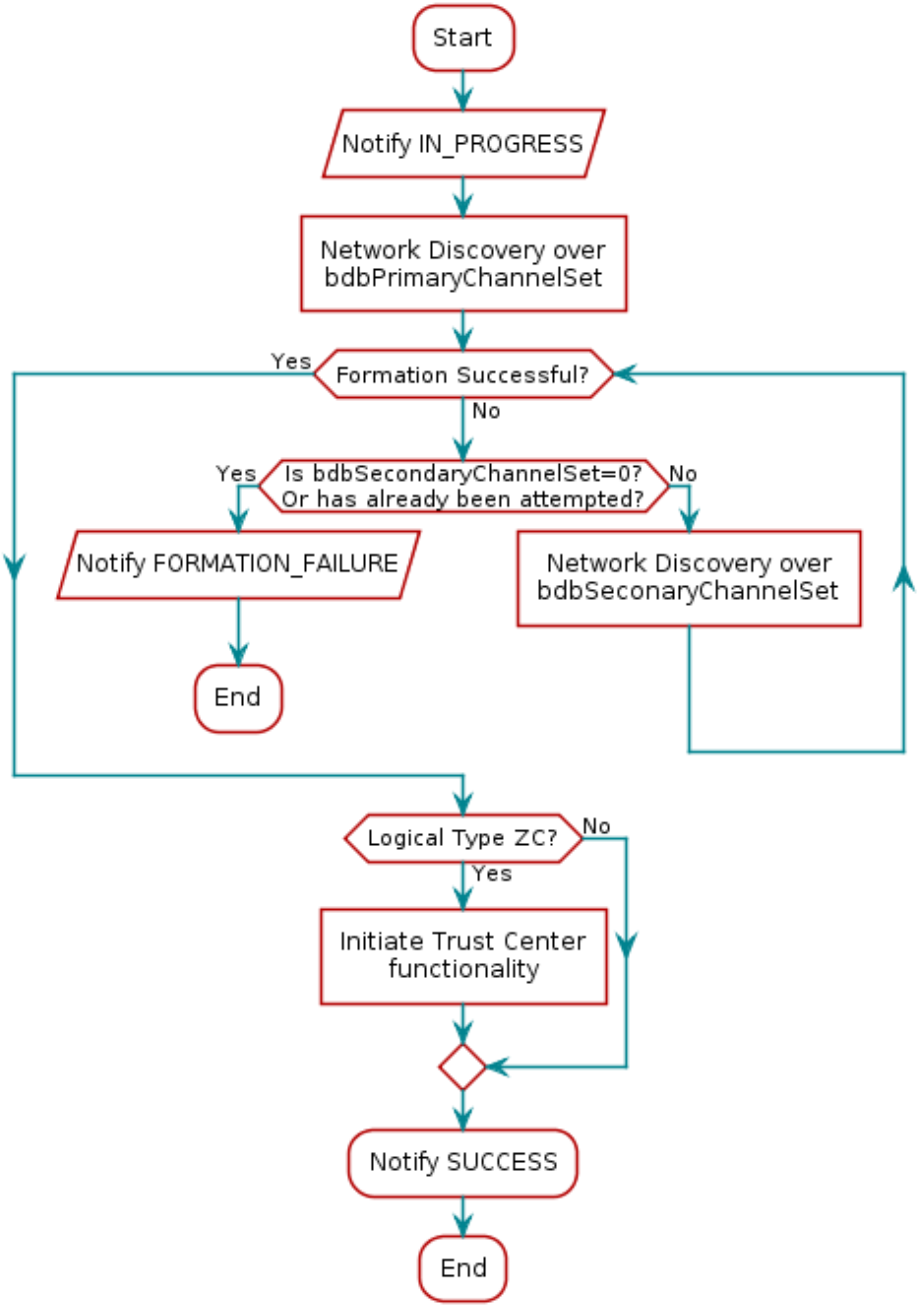

Figure 68. Network Formation

Finding and Binding

The Finding and Binding procedure can be performed as initiator, target, or both, depending on the clusters of the endpoint performing the Finding and Binding procedure. For example, 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 7 Specification [2].

The application must specify with which endpoint it wants to perform the finding and binding procedure by calling `Zstackapi_bdbSetIdentifyActiveEndpointReq()`. Note that the endpoint indicated must contain the Identify cluster in order to perform the procedure.

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 procedure sends a `BDB_COMMISSIONING_FB_INITIATOR_IN_PROGRESS` notification when it starts and a zstackmsg `zstackmsg_CmdIDs_BDB_IDENTIFY_TIME_CB` when it finishes.

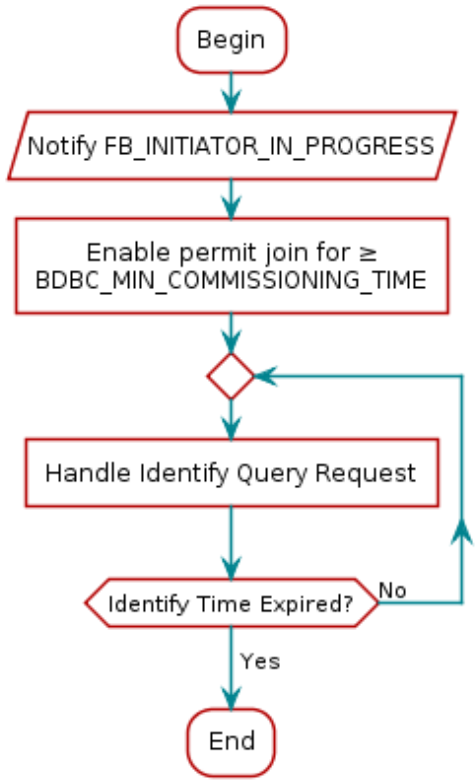

Figure 69. Finding & Binding Procedure for a Target Endpoint

Finding and Binding Procedure for an Initiator Endpoint

In this procedure, the initiator will search for identifying endpoints by broadcasting identify query commands 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 the target endpoints.

The finding and binding process for an initiator device is illustrated in Figure 70. and described here:

1. The application is notified about the commissioning method starting, and the local device broadcasts 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 (respondent devices).

2. The local device sends a ZDO simple descriptor request to each respondent device one at a time. If no response is received, the local device will send a ZDO simple

descriptor request to the next respondent in the list. If no response from a respondent is received after FINDING_AND_BINDING_MAX_ATTEMPTS attempts, the local device marks that respondent as completely processed. This step repeats until all respondents are completely processed.

3. Upon the reception of a simple descriptor response, the local device will search for application clusters compatible with the endpoint in the local device that is performing the Finding and Binding procedure.
4. If the device is performing unicast binds (`BDB_DEFAULT_COMMISSIONING_GROUP_ID == 0xFFFF`), it searches for the IEEE address of the respondent device in the address manager. If not found, it sends a ZDO IEEE address request command. The device reattempts this process until the IEEE address response is received from the respondent and the bind entry is created for the matching clusters. After FINDING_AND_BINDING_MAX_ATTEMPTS attempts, this respondent is marked as processed without adding any bind. Group binds are created if any match is found. For the definition of application clusters, refer to the Zigbee ZCL 7 Specification [2].
   a. The application will receive bind notifications via a zstackmsg of ID `zstackmsg_CmdIDs_BDB_BIND_NOTIFICATION_CB` .
   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.
5. The local device will repeat steps 2 to 4 until all respondents are marked as processed. Then it will send a `BDB_COMMISSIONING_SUCCESS` notification to the application.

The Finding and Binding procedure for groups enables APS Acknowledgements to increase reliability of creating the group membership at the remote device.

The finding and binding process for an initiator device can be configured to perform periodically every `FINDING_AND_BINDING_PERIODIC_TIME` seconds instead of `BDBC_MIN_COMMISSIONING_TIME` (180) seconds. This configuration is determined by `FINDING_AND_BINDING_PERIODIC_ENABLE` (default value is `TRUE`). In this case, multiple identify query command responses from the same device will not be duplicated in the respondent list. The Finding and Binding process can be terminated early by calling `Zstackapi_bdbStopInitiatorFindingBindingReq()` .

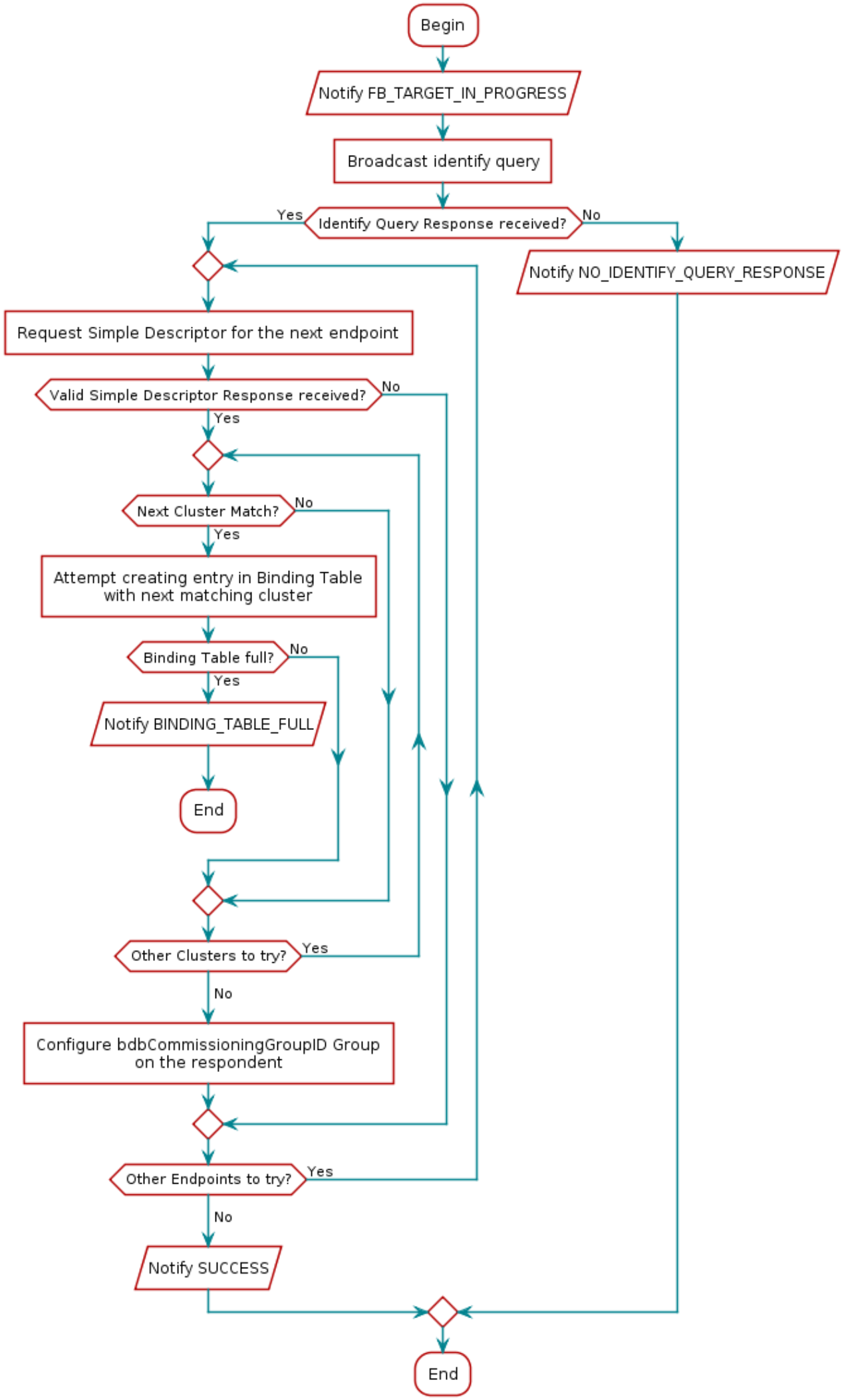
Begin
Notify FB_TARGET_IN_PROGRESS
Broadcast identify query
Identify Query Response received?
Yes
No
Notify NO_IDENTIFY_QUERY_RESPONSE
Request Simple Descriptor for the next endpoint
Valid Simple Descriptor Response received?
No
Yes
Next Cluster Match?
No
Yes
Attempt creating entry in Binding Table with next matching cluster
Binding Table full?
No
Yes
Notify BINDING_TABLE_FULL
End
Other Clusters to try?
Yes
No
Configure bdbCommissioningGroupID Group on the respondent
Other Endpoints to try?
Yes
No
Notify SUCCESS
End

Figure 70. Finding & Binding Procedure for an Initiator Endpoint

Touchlink Commissioning

Touchlink commissioning is an optional commissioning mechanism defined in the Zigbee BDB specification, where nodes are commissioned using inter-PAN communication. It requires physical proximity between devices.

Configurations

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

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 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` and `TOUCHLINK_CERTIFICATION_LINK_KEY` with the actual secret values.

2. Change the `TOUCHLINK_KEY_INDEX` definition to `TOUCHLINK_KEY_INDEX_MASTER`.

Constants

The BDB defines constants and internal attribute defaults to allow a device to manage how the Touchlink device operates (see section 5.2 in Base Device Behavior Specification [3]).

Definition	Specification's Constant / Attribute default	Va
BDBCTL_INTER_PAN_TRANS_ID_LIFETIME	*bdbcTLInterPANTransIdLifetime*	80
BDBCTL_MIN_STARTUP_DELAY_TIME	*bdbcTLMinStartupDelayTime*	20
BDBCTL_PRIMARY_CHANNEL_LIST	*bdbcTLPrimaryChannelSet*	0x
BDBCTL_RX_WINDOW_DURATION	*bdbcTLRxWindowDuration*	50
BDBCTL_SCAN_TIME_BASE_DURATION	*bdbcTLScanTimeBaseDuration*	25

Definition	Specification's Constant / Attribute default	Va
BDBCTL_SECONDARY_CHANNEL_LIST	*bdbcTLSecondaryChannelSet*	0x

Definitions Derived From the Zigbee Base Device Behavior Specification

Endpoint Setup

Since the Touchlink commissioning is managed by a dedicated task separate from the applications, its endpoint and device ID may be re-defined. The endpoint must be a valid value that is not used by the device's other endpoints. The device ID must not equal any valid device ID (to prevent accidental matches).

`TOUCHLINK_INTERNAL_ENDPOINT` (default = 13).

`TOUCHLINK_INTERNAL_DEVICE_ID` (default = 0xE15E).

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:

`BDB_TL_IDENTIFY_TIME`

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 [3]), 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 atouch-link failure due to transaction lifetime expiration, especially if done on the secondary channel set.

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).

Application Selective Target Touch-Link

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 targets 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.).

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.

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, which will allow testing of multiple Touchlink devices set in the same space without interference. This should be used for testing purposes only. `TOUCHLINK_CH_OFFSET` is defined by default as `No_Ch_offset`, which means that no shift is applied to the primary channel set.

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.

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.

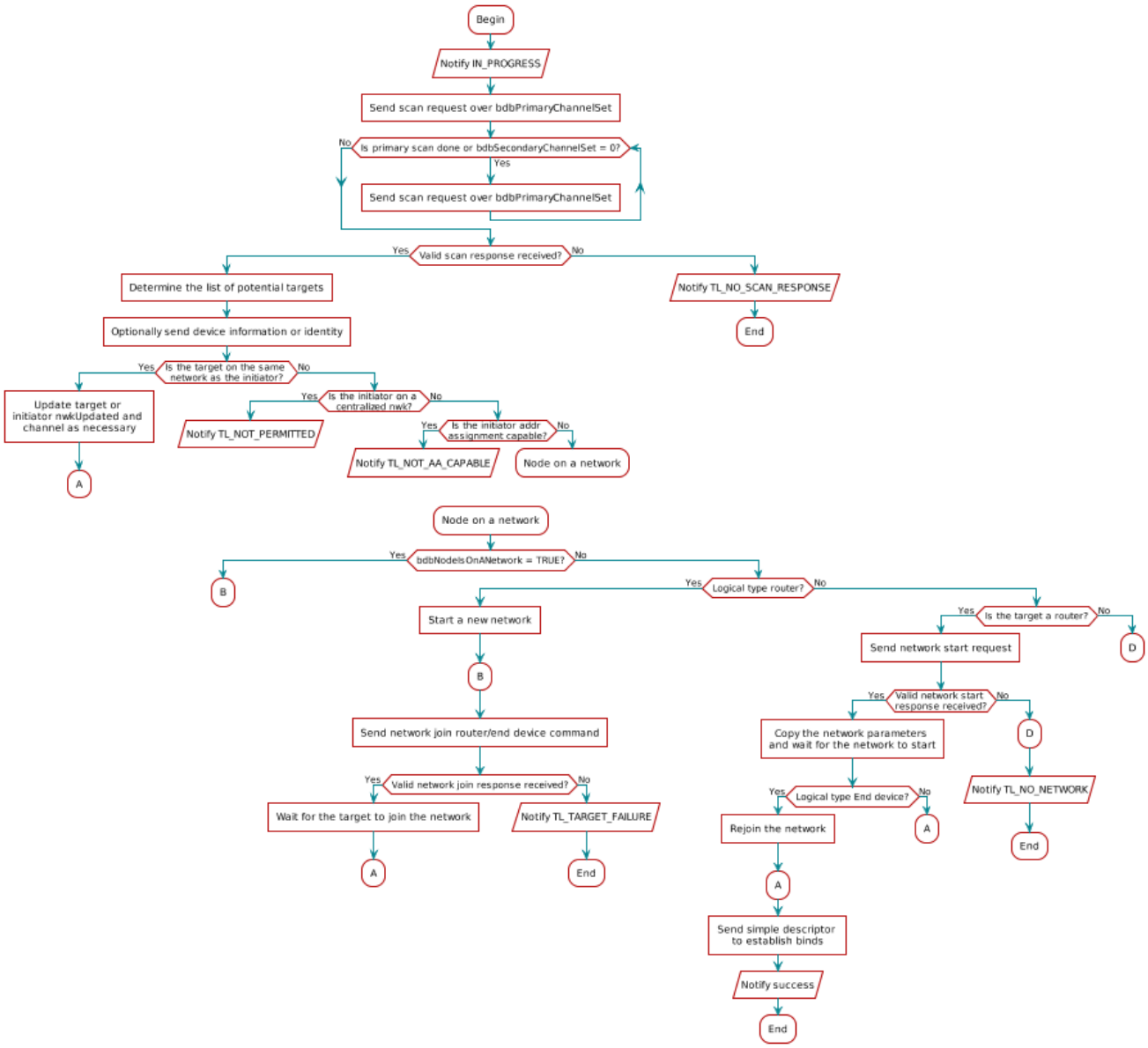

Figure 71. Touchlink Commissioning Procedure for an Initiator

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 to join the initiator network.

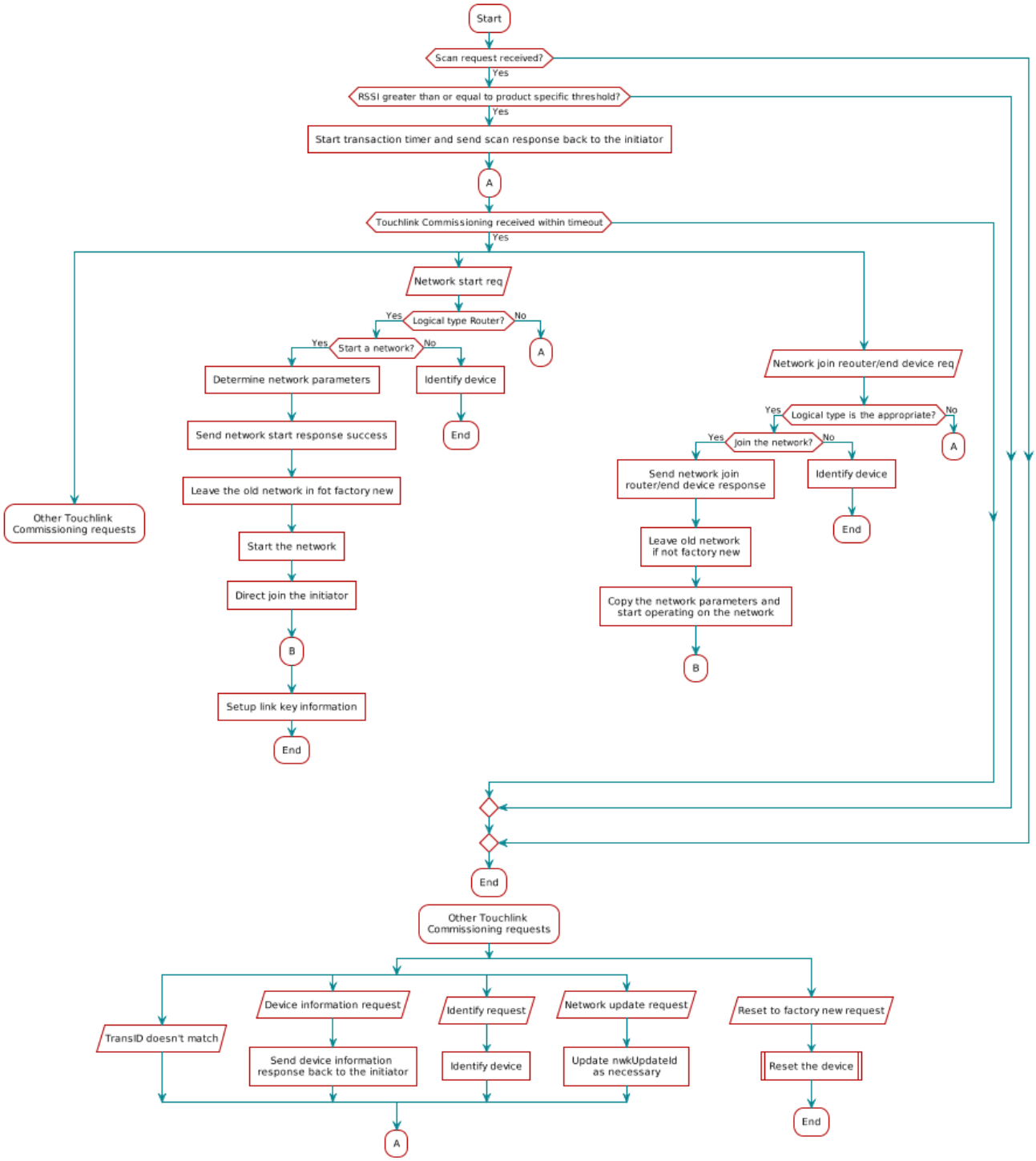

Figure 72. Touchlink Commissioning Procedure for a Target

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:

Reset via Basic Cluster

If the application's Basic Cluster supports reset commands, then the application must reset all attributes in every cluster supported by the device. This command must not affect any network parameters, binds, or groups. The application implements this in the callback function for this command.

Reset via Touchlink Commissioning Cluster

If touchlink as target is supported, then this reset mechanism will trigger the device to issue a leave request for itself with *Rejoin* set to `FALSE` and *RemoveChildren* set to `FALSE`. See Reset via Network Leave Request for further details on the leave request.

Reset via Mgmt_leave_req ZDO command

If the command is valid, then the receiving device will issue a leave request for itself with *Rejoin* set to `FALSE` and *RemoveChildren* set to `FALSE`. See Reset via Network Leave Request for further details on the leave request.

Reset via Local Action

This type of reset is triggered by the user (eg. a button press). Non-Coordinator devices will issue a network leave request for itself with *Rejoin* set to `FALSE` and *RemoveChildren* set to `FALSE`. Coordinator devices will clear persistent Zigbee data (since they cannot process network leave commands). See Reset via Network Leave Request for further details on leave requests. The application may trigger this reset by calling `Zstackapi_bdbResetLocalActionReq()`.

Reset via Network Leave Request

Network leave requests are processed differently across device types. Coordinator devices ignore the command (including those it issues for itself). Router devices process leave requests issued by itself and any other device in the network (if `zgNwkLeaveRequestAllowed` is enabled). End devices only accept leave requests issued by itself and or its parent device. A valid request will cause the device to clear all persistent Zigbee data (bindings, network parameters, groups, attributes, etc.) except the outgoing network frame counter for the network it is leaving.

Network Manager

Overview

A single device can become the Network Manager. This device is responsible for receiving channel interference reports and PAN ID conflict reports. Based on these reports, the Network Manager changes the network channel and PAN ID as needed.

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 `zd_nwk_mgr.c` and `zd_nwkmgr.h` files.

Channel Interference

The Network Manager implements frequency agility measures if interference is detected. This section explains how the channel of a network can be changed using `Mgmt_NWK_Update_req` and `Mgmt_NWK_Update_notify` commands.

Channel Interference Detection

The coordinator and all routers transmit failures using the Transmit Failure field in their neighbor tables. They also keep a NIB counter for total transmissions attempted. If this counter exceeds `ZDNWKMGR_MIN_TRANSMISSIONS` (20) and the number of transmit failure is greater than `ZDNWKMGR_CI_TX_FAILURE` (25) percent of the messages sent, the device may have detected interference on the current channel.

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 that 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.

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, then 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 lowest energy in the `Mgmt_NWK_Update_notify` message. This is the proposed new channel. If this new channel does not have an energy level below an acceptable threshold `ZDNWKMGR_ACCEPTABLE_ENERGY_LEVEL`, then a channel change should not be considered.

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 prepares to broadcast (to the coordinator and all routers) a `Mgmt_NWK_Update_req`, which will notify devices of the new channel. It increments the `nwkUpdateId` parameter in the NIB and beacon payload, and includes it in the `Mgmt_NWK_Update_req`. After broadcasting, the Network Manager sets up a timer lasting `ZDNWKMGR_UPDATE_REQUEST_TIMER` seconds. Another `Mgmt_NWK_Update_req` to change channels will not be issued before 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 and 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.

Quick Reference

Setting minimum transmissions attempted for Channel Interference detection	Set `ZDNWKMGR_MI`
Setting minimum transmit failure rate for Channel Interference detection	Set `ZDNWKMGR_CI`

Setting minimum transmit failure rate for Channel Change	Set `ZDNWKMGR_CC`
Setting acceptable energy level threshold for Channel Change	Set `ZDNWKMGR_AC`
Setting APS channel timer for issuing Channel Changes	Set `ZDNWKMGR_UP`

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 the PAN ID of a network can be updated using the Network Report and Update commands.

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.

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.

Green Power

Introduction

As a requirement for Z3.0 certification, all Zigbee routing devices (coordinators and routers) must support the Green Power Basic proxy, which is an application that can relay commands from a Green Power Device (GPD) to a Green Power Sink device.

A GPD has very limited power or relies on energy harvesting to function. It cannot perform the two way communication for establishing association to a Zigbee network. Instead, 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 depend on its capabilities and resources. The details of those commissioning methods and commands are beyond 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 is defined in such a way that different applications can sit on top of it, such as a Sink Device. For further details on Sink Device implementation refer to [4].

GP is implemented in the Zigbee reserved endpoint 242.

Green Power Basic Proxy

Since the GP basic proxy is an application to relay the commands to a Sink device, no functionality needs to be handled by the application (e.g. light, switch, etc.) that's running the GP basic proxy. The only interface between them is the following:

- `gp_RegisterGPChangeChannelReqCB()`: Register a callback that requests for permission to switch the operational channel to the GPD's channel to perform commissioning. The callback registered may return `FALSE` to not allow the channel change if an application operation cannot be interrupted. Permission is also requested from the BDB module. If the callback returns `TRUE` or no callback is registered, then the GP basic proxy application will handle the change of channels. The channel change lasts for at most `gpBirectionalCommissioningChangeChannelTimeout` (5 seconds).

Green Power Sink

The GP Sink is a service that the device application can register for, which allows the device to receive and process GP Data Frames. A callback list is provided for the device application to handle GP notification commands.

```
typedef struct
{
#ifdef ZCL_IDENTIFY
  GPDFCB_GP_identify_t                pfnGpdfIdentifyCmd;
#endif
#ifdef ZCL_SCENES
  GPDFCB_GP_RecallScene_t             pfnGpdfRecallSceneCmd;
  GPDFCB_GP_StoreScene_t              pfnGpdfStoreSceneCmd;
#endif
#ifdef ZCL_ON_OFF
  GPDFCB_GP_Off_t                     pfnGpdfOffCmd;
  GPDFCB_GP_On_t                      pfnGpdfOnCmd;
  GPDFCB_GP_Toggle_t                  pfnGpdfToggleCmd;
#endif
#ifdef ZCL_LEVEL_CTRL
  GPDFCB_GP_LevelControlStop_t        pfnGpdfLevelControlStopCmd;
  GPDFCB_GP_MoveUp_t                  pfnGpdfMoveUpCmd;
  GPDFCB_GP_MoveDown_t                pfnGpdfMoveDownCmd;
  GPDFCB_GP_StepUp_t                  pfnGpdfStepUpCmd;
  GPDFCB_GP_StepDown_t                pfnGpdfStepDownCmd;
  GPDFCB_GP_MoveUpWithOnOff_t         pfnGpdfMoveUpWithOnOffCmd;
  GPDFCB_GP_MoveDownWithOnOff_t       pfnGpdfMoveDownWithOnOffCmd;
  GPDFCB_GP_StepUpWithOnOff_t         pfnGpdfStepUpWithOnOffCmd;
  GPDFCB_GP_StepDownWithOnOff_t       pfnGpdfStepDownWithOnOffCmd;
#endif
  GPDFCB_GP_MoveHueStop_t             pfnGpdfMoveHueStopCmd;
  GPDFCB_GP_MoveHueUp_t               pfnGpdfMoveHueUpCmd;
  GPDFCB_GP_MoveHueDown_t             pfnGpdfMoveHueDownCmd;
  GPDFCB_GP_StepHueUp_t               pfnGpdfStepHueUpCmd;
  GPDFCB_GP_StepHueDown_t             pfnGpdfStepHueDownCmd;
  GPDFCB_GP_MoveSaturationStop_t      pfnGpdfMoveSaturationStopCmd;
  GPDFCB_GP_MoveSaturationUp_t        pfnGpdfMoveSaturationUpCmd;
  GPDFCB_GP_MoveSaturationDown_t      pfnGpdfMoveSaturationDownCmd;
  GPDFCB_GP_StepSaturationUp_t        pfnGpdfStepSaturationUpCmd;
  GPDFCB_GP_StepSaturationDown_t      pfnGpdfStepSaturationDownCmd;
  GPDFCB_GP_MoveColor_t               pfnGpdfMoveColorCmd;
  GPDFCB_GP_StepColor_t               pfnGpdfStepColorCmd;
#ifdef ZCL_DOORLOCK
  GPDFCB_GP_LockDoor_t                pfnGpdfLockDoorCmd;
  GPDFCB_GP_UnlockDoor_t              pfnGpdfUnlockDoorCmd;
#endif
  GPDFCB_GP_AttributeReporting_t      pfnGpdfAttributeReportingCmd;
  GPDFCB_GP_MfrSpecificReporting_t    pfnGpdfMfrSpecificReportingCmd;
  GPDFCB_GP_MultiClusterReporting_t   pfnGpdfMultiClusterReportingCmd;
  GPDFCB_GP_MfrSpecificMultiReporting_t pfnGpdfMfrSpecificMultiReportingCmd;
  GPDFCB_GP_RequestAttributes_t       pfnGpdfRequestAttributesCmd;
  GPDFCB_GP_ReadAttributeRsp_t        pfnGpdfReadAttributeRspCmd;
  GPDFCB_GP_zclTunneling_t            pfnGpdfzclTunnelingCmd;
} GpSink_AppCallbacks_t;
```

`zclGp_RegisterCBForGPDCommand()` allows to register the callback list with the user defined callback functions. When a notification for a registered command functions arrives the Sink will relay the frame to the application callback.

Inter-PAN Transmission

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.

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 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 than the device's NIB channel.
- 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.
- 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:

1. Calls `StubAPS_AppRegister()` API to register itself with the Stub APS layer.

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

Quick Reference

Setup application as InterPAN application.	Call `StubAPS_RegisterApp(app_enc`
Set InterPAN channel.	Call `StubAPS_SetInterPanChannel(`
Send InterPAN Message.	Call `AF_DataRequest()` with: • `dstPanID` different from • `dst address endpoint == STU`
Receive an InterPAN message.	Receive an `AF_INCOMING_MSG_CMD`
End the InterPAN session by putting back the IntraPAN channel.	Call `StubAPS\_SetIntraPanChannel`

ZMAC LQI Adjustment

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.

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()`.

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 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`. Alternate values for these parameters can be provided as compiler directives in the IDE project file or in one of Z-Stack's configuration files (`zstack_config.h`, `f8wCoord.opts`, etc.). Refer to the radio's data sheet for information on the normal minimum/maximum correlation values.