Freescale BeeStack™

Software Reference Manual

Document Number: BSSRM
Rev. 1.0
07/2007

How to Reach Us:

Home Page:
www.freescale.com

E-mail:
support@freescale.com

USA/Europe or Locations Not Listed:
Freescale Semiconductor
Technical Information Center, CH370
1300 N. Alma School Road
Chandler, Arizona 85224
+1-800-521-6274 or +1-480-768-2130
support@freescale.com

Europe, Middle East, and Africa:
Freescale Halbleiter Deutschland GmbH
Technical Information Center
Schatzbogen 7
81829 Muenchen, Germany
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)
support@freescale.com

Japan:
Freescale Semiconductor Japan Ltd.
Headquarters
ARCO Tower 15F
1-8-1, Shimo-Meguro, Meguro-ku,
Tokyo 153-0064, Japan
0120 191014 or +81 3 5437 9125
support.japan@freescale.com

Asia/Pacific:
Freescale Semiconductor Hong Kong Ltd.
Technical Information Center
2 Dai King Street
Tai Po Industrial Estate
Tai Po, N.T., Hong Kong
+800 2666 8080
support.asia@freescale.com

***For Literature Requests Only*:**
Freescale Semiconductor Literature Distribution Center
P.O. Box 5405
Denver, Colorado 80217
1-800-521-6274 or 303-675-2140
Fax: 303-675-2150
LDCForFreescaleSemiconductor@hibbertgroup.com

Information in this document is provided solely to enable system and software implementers to use Freescale Semiconductor products. There are no express or implied copyright licenses granted hereunder to design or fabricate any integrated circuits or integrated circuits based on the information in this document.

Freescale Semiconductor reserves the right to make changes without further notice to any products herein. Freescale Semiconductor makes no warranty, representation or guarantee regarding the suitability of its products for any particular purpose, nor does Freescale Semiconductor assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. “Typical” parameters that may be provided in Freescale Semiconductor data sheets and/or specifications can and do vary in different applications and actual performance may vary over time. All operating parameters, including “Typicals”, must be validated for each customer application by customer’s technical experts. Freescale Semiconductor does not convey any license under its patent rights nor the rights of others. Freescale Semiconductor products are not designed, intended, or authorized for use as components in systems intended for surgical implant into the body, or other applications intended to support or sustain life, or for any other application in which the failure of the Freescale Semiconductor product could create a situation where personal injury or death may occur. Should Buyer purchase or use Freescale Semiconductor products for any such unintended or unauthorized application, Buyer shall indemnify and hold Freescale Semiconductor and its officers, employees, subsidiaries, affiliates, and distributors harmless against all claims, costs, damages, and expenses, and reasonable attorney fees arising out of, directly or indirectly, any claim of personal injury or death associated with such unintended or unauthorized use, even if such claim alleges that Freescale Semiconductor was negligent regarding the design or manufacture of the part.

Freescale™ and the Freescale logo are trademarks of Freescale Semiconductor, Inc. All other product or service names are the property of their respective owners.

© Freescale Semiconductor, Inc. 2006, 2007. All rights reserved.

Contents

About This Book . . . vii
Audience . . . vii
Organization . . . vii
Revision History . . . viii
Conventions . . . viii
Definitions, Acronyms, and Abbreviations . . . viii
Reference Materials . . . x

Chapter 1
Introduction

1.1 What This Document Describes . . . 1-1
1.2 What This Document Does Not Describe . . . 1-1

Chapter 2
ZigBee Overview

2.1 Network Elements . . . 2-2
2.1.1 Device Types . . . 2-2
2.1.2 Star Network . . . 2-4
2.1.3 Tree Network . . . 2-5
2.1.4 Mesh Network . . . 2-6
2.1.5 Personal Area Network . . . 2-7
2.1.6 Channels . . . 2-7
2.1.7 Device and Service Discovery . . . 2-7
2.1.8 Addressing/Messaging . . . 2-7
2.1.9 Binding . . . 2-8
2.2 Application Elements . . . 2-9
2.2.1 Applications . . . 2-9
2.2.2 Attributes . . . 2-10
2.2.3 Clusters . . . 2-10
2.2.4 Endpoints . . . 2-10

Chapter 3
BeeStack Features

3.1 BeeStack Task Scheduler . . . 3-1
3.2 BeeStack Application Programming Interface . . . 3-2
3.3 Source Files – Directory Structure . . . 3-4
3.4 Miscellaneous Source Files . . . 3-5

Chapter 4
Application Framework

4.1 AF Types . . . 4-2
4.2 Endpoint Management . . . 4-3

4.2.1 Simple Descriptor 4-3
4.2.2 Register Endpoint 4-3
4.2.3 De-register Endpoint 4-4
4.2.4 Get Endpoint 4-4
4.2.5 Find Endpoint Descriptor 4-4
4.3 Message Allocation. 4-5
4.4 AF Data Requests 4-5
4.5 AF Data Indications 4-8

Chapter 5
Application Support Sub-layer

5.1 Direct and Indirect Data Addressing 5-2
5.2 APS Layer Interface 5-2
5.2.1 Get Request. 5-2
5.2.2 Set Request 5-3
5.2.3 Get Table Entry. 5-3
5.2.4 Set Table Entry 5-3
5.2.5 Add to Address Map 5-3
5.2.6 Remove from Address Map 5-4
5.2.7 Find IEEE Address in Address Map. 5-4
5.2.8 Get NWK Address from IEEE Address 5-4
5.2.9 Get IEEE Address from NWK Address 5-4
5.3 Binding 5-5
5.3.1 Bind Request. 5-5
5.3.2 Unbind Request. 5-5
5.3.3 Find Binding Entry 5-6
5.3.4 Find Next Binding Entry. 5-6
5.3.5 Clear Binding Table 5-6
5.3.6 Add Group Request 5-7
5.3.7 Remove Group Request 5-7
5.3.8 Remove Endpoint from All Groups Request 5-7
5.3.9 Identify Endpoint Group Membership 5-8
5.3.10 Group Reset Function 5-8
5.4 AIB Attributes. 5-8

Chapter 6
ZigBee Device Objects

6.1 ZDO State Machine 6-2
6.2 General ZDO Interfaces 6-2
6.2.1 Get State Machine. 6-2
6.2.2 Start ZDO State Machine without NVM 6-3
6.2.3 Start ZDO State Machine with NVM 6-3
6.2.4 Stop ZDO State Machine 6-3
6.2.5 Stop ZDO and Leave 6-3

6.3 Device Specific ZDO Interfaces 6-4
6.3.1 ZC State Machine 6-4
6.3.2 ZR State Machine 6-5
6.3.3 ZED Machine State 6-6
6.4 Selecting PAN ID, Channel and Parent 6-8

Chapter 7
ZigBee Device Profile

7.1 Application Support Layer 7-1
7.2 Device and Service Discovery 7-2
7.2.1 Device Discovery 7-2
7.2.2 Service Discovery 7-2
7.3 Primary Discovery Cache Device Operation 7-3
7.4 Binding Services 7-4
7.5 ZDP Functions and Macros 7-4
7.5.1 ZDP Register Callback 7-4
7.5.2 ZDP NLME Synchronization Request 7-4
7.6 Device and Service Discovery – Client Services 7-5
7.6.1 Network Address Request 7-6
7.6.2 IEEE Address Request Command 7-6
7.6.3 Node Descriptor Request 7-6
7.6.4 Power Descriptor Request 7-7
7.6.5 Simple Descriptor Request 7-7
7.6.6 Active Endpoint Request 7-7
7.6.7 Match Descriptor Request 7-8
7.6.8 Complex Descriptor Request 7-8
7.6.9 User Descriptor Request 7-8
7.6.10 Discovery Cache Request 7-9
7.6.11 End Device Announce 7-9
7.6.12 User Descriptor Set Request 7-10
7.6.13 Server Discovery Request 7-10
7.6.14 Discovery Cache Storage Request 7-10
7.6.15 Store Node Descriptor on Primary Cache Device 7-11
7.6.16 Store Power Descriptor Request 7-11
7.6.17 Active Endpoint List Storage Request 7-12
7.6.18 Simple Descriptor Storage Request 7-12
7.6.19 Remove Node Cache Request 7-13
7.6.20 Find Node Cache Request 7-13
7.7 Binding Management Service Commands 7-13
7.7.1 End Device Bind Request 7-14
7.7.2 Bind Request 7-15
7.7.3 Unbind Request 7-15
7.7.4 Local Bind Register Request 7-16
7.7.5 Replace Device Request 7-16

7.7.6 Store Backup Bind Entry Request 7-17
7.7.7 Remove Entry from Backup Storage 7-17
7.7.8 Backup Binding Table Request 7-18
7.7.9 Recover Binding Table Request 7-18
7.7.10 Source Binding Table Backup Request 7-19
7.7.11 Recover Source Binding Table Request 7-19
7.8 Network Management Services 7-20
7.8.1 Management Network Discovery Request 7-20
7.8.2 Management LQI Request 7-20
7.8.3 Routing Discovery Management Request 7-21
7.8.4 Management Bind Request 7-21
7.8.5 Management Leave Request 7-21
7.8.6 Management Permit Joining 7-22
7.8.7 Management Cache 7-22
7.9 ZDO Layer Status Values 7-22

Chapter 8
Network Layer

8.1 Channel and PAN Configuration 8-2
8.1.1 Channel Configuration 8-2
8.1.2 PAN ID 8-3
8.1.3 Beacon Notify 8-3
8.1.4 NWK Layer Interfaces 8-4
8.1.5 NWK Layer Filters 8-5
8.2 NWK Information Base 8-5

Chapter 9
Application Support Layer

9.1 ASL Utility Functions 9-1
9.2 ASL Data Types 9-1
9.3 ASL Utility Functions 9-3
9.3.1 Initialize User Interface 9-3
9.3.2 Set Serial LEDs 9-3
9.3.3 Stop Serial LEDs 9-3
9.3.4 Set LED State 9-3
9.3.5 Write to LCD 9-3
9.3.6 Change User Interface Mode 9-4
9.3.7 Display Current User Interface Mode 9-4
9.3.8 Update Device 9-4
9.3.9 Handle Keys 9-4
9.3.10 Display Temperature 9-4

Chapter 10

BeeStack Common Functions

10.1 BeeStack Common Prototypes 10-1
10.2 Common Network Functions 10-2

Chapter 11
User-Configurable BeeStack Options

11.1 Compile-Time Options 11-1
11.2 More Compile-time Options 11-3

Chapter 12
BeeStack Security

12.1 Security Overview 12-1
12.2 Security Configuration Properties 12-2
12.2.1 mDefaultValueOfNwkKeyPreconfigured_c. 12-2
12.2.2 mDefaultValueOfNwkSecurityLevel_c 12-2
12.2.3 mDefaultValueOfNetworkKey_c 12-2
12.2.4 gDefaultValueOfMaxEntriesForExclusionTable_c 12-2
12.3 ZigBee Trust Center Authentication. 12-3

About This Book

This manual describes BeeStack, the Freescale implementation of the ZigBee wireless network protocol stack. This manual explains the standard interfaces and device definitions that permit interoperability among ZigBee devices.

Audience

This document is intended for software developers who write applications for BeeStack-based products using Freescale development tools. It describes BeeStack APIs, control features, code examples, and functional variables.

Organization

This document is organized into the following sections.

Chapter 1	Introduction – describes this document.
Chapter 2	ZigBee Overview – introduces ZigBee network concepts.
Chapter 3	BeeStack Overview – introduces the BeeStack architecture and source file structure.
Chapter 4	Application Framework – introduces the function calls, macros, and APIs available in the Application Framework (AF).
Chapter 5	Application Support Sub-layer – describes the function calls, macros, and APIs available in the Application Support Sub-layer (APS).
Chapter 6	ZigBee Device Objects – introduces the function calls, macros, and APIs available in the ZigBee device objects (ZDO).
Chapter 7	ZigBee Device Profile – introduces the ZigBee device profile (ZDP) and associated macros, function calls, and prototypes.
Chapter 8	Network Layer – describes the function calls and macros available in the network (NWK) layer.
Chapter 9	Application Support Layer – introduces the Application support functions and macros.
Chapter 10	BeeStack Common Functions – introduces the BeeStack common interface macros and function calls.
Chapter 11	User-Configurable BeeStack Options – introduces the BeeStack configurable items.
Chapter 12	BeeStack Security – describes how BeeStack supports full ZigBee security for stack profile 0x01 of the ZigBee 2006 specification.

Revision History

The following table summarizes revisions to this document since the previous release (Rev. 0.0).

Revision History

Location	Revision
Section 3.4	New Section
Section 4.2.2	Clarified last sentence.
Deleted 4.2.3	Removed initiate data request.
Rewrote 4.2.4	Rewrote get endpoint.
Rewrote 4.3	Rewrote message allocation.
Section 5.2	Added text and edited various sections.
Section 6.2	Added text and edited various sections.
Section 7.1	Added text and edited various sections.
Section 7.5.1	Rewrote section.
Section 7.6	Rewrote various sections.
Section 8.3	Edited for clarity.
Section 9.1 and 9.2	Added information.
Table 11-1	Added information.
Table 11-2	Added information.

Conventions

This *BeeStack Reference Manual* uses the following formatting conventions when detailing commands, parameters, and sample code:

`Courier mono-space` type indicates commands, command parameters, and code examples.

Bold style indicates the command line elements, which must be entered exactly as written.

Italic type indicates command parameters that the user must type in or replace, as well as emphasizes concepts or foreign phrases and words.

Definitions, Acronyms, and Abbreviations

Acronym or Term	Definition
ACK	Acknowledgement
ACL	Access control list
AF	Application framework
AIB	Application support sub-layer information base
APDU	Application support sub-layer protocol data unit
API	Application programming interface
APL	Application layer
APS	Application support sub-layer
APSDE	APS data entity
APSDE-SAP	APS data entity - service access point
APSME	APS management entity
APSME-SAP	APS management entity - service access point

ASDU	APS service data unit
OTA	Over the air: a radio frequency transmission
Binding	Matching ZigBee devices based on services and needs
BTR	Broadcast transaction record, the local receipt of a broadcast message
BTT	Broadcast transaction table, holds all BTRs
CBC-MAC	Cipher block chaining message authentication code
CCA	Clear channel assessment
Cluster	A collection of attributes associated with a specific cluster-identifier
Cluster identifier	An enumeration that uniquely identifies a cluster within an application profile
CSMA-CA	Carrier sense multiple access with collision avoidance
CTR	Counter
Data Transaction	Process of data transmission from the endpoint of a sending device to the endpoint of the receiving device
Device/Node	ZigBee network component containing a single IEEE 802.15.4 radio
Direct addressing	Direct data transmission including both destination and source endpoint fields
Endpoint	Component within a unit; a single IEEE 802.15.4 radio may support up to 240 independent endpoints
IB	Information base, the collection of variables configuring certain behaviors in a layer
IEEE	Institute of Electrical and Electronics Engineers, a standards body
Indirect addressing	Transmission including only the source endpoint addressing field along with the indirect addressing bit
ISO	International Standards Organization
LCD	Liquid crystal display
LED	Light-emitting diode
LQI	Link quality indicator or indication
MAC	Medium access control sub-layer
MCPS-SAP	MAC common part sub-layer - service access point
MIC	Message integrity code
MLME	MAC sub-layer management entity
MLME-SAP	MAC sub-layer management entity service access point
NIB	Network layer information base
NLDE	Network layer data entity
NLDE-SAP	Network layer data entity - service access point
NLME	Network layer management entity
NLME-SAP	Network layer management entity - service access point

NPDU	Network protocol data unit
NSDU	Network service data unit
NVM	Non-volatile memory
NWK	Network layer
Octet	Eight bits of data, or one byte
OSI	Open System Interconnect
PAN	Personal area network
PD-SAP	Physical layer data - service access point
PDU	Protocol data unit (packet)
PHY	Physical layer
PIB	Personal area network information base
PLME-SAP	Physical layer management entity - service access point
Profile	Set of options in a stack or an application
RF	Radio frequency
SAP	Service access point
SKG	Secret key generation
SKKE	Symmetric-key key establishment protocol
SSP	Security service provider, a ZigBee stack component
Stack	ZigBee protocol stack
WDA	wireless demo application
WPAN	wireless personal area network
ZDO	ZigBee device object(s)
ZDP	ZigBee device profile
802.15.4	An IEEE standard radio specification that underlies the ZigBee Specification

Reference Materials

This following served as references for this manual:

1. Document 053474r13, *ZigBee Specification*, ZigBee Alliance, December 2006
2. Document 06027r04, *ZB_AFG-ZCL_Foundation*, ZigBee Alliance, October 2006
3. Document 053520r16, *ZB_HA_PTG-Home-Automation-Profile*, ZigBee Alliance, September 2006

Chapter 1
Introduction

This manual describes the Freescale BeeStack protocol stack, its components, and their functional roles in building wireless networks. The function calls, application programming interfaces (API), and code examples included in this manual address every component required for communication in a ZigBee wireless network.

1.1 What This Document Describes

This manual provides ZigBee software designers and developers all of the function prototypes, macros, and stack libraries required to develop applications for ZigBee wireless networks.

1.2 What This Document Does Not Describe

This manual does not describe how to install software, configure the hardware, or set up and use ZigBee applications.

See the following documents for help in setting up the Freescale hardware and using other Freescale software to configure devices.

- *Freescale ZigBee Applications User's Guide*, (ZAUG)
- *Freescale BeeKit Wireless Connectivity Toolkit User's Guide*, (BKWCTKUG)

Chapter 2
ZigBee Overview

The BeeStack architecture builds on the ZigBee protocol stack. Based on the OSI Seven-Layer model, the ZigBee stack ensures interoperability among networked devices. The physical (PHY), media access control (MAC), and network (NWK) layers create the foundation for the application (APL) layers.

BeeStack defines additional services to improve the communication between layers of the protocol stack. At the Application Layer, the application support layer (ASL) facilitates information exchange between the Application Support Sub-Layer (APS) and application objects. Finally, ZigBee Device Objects (ZDO), in addition to other manufacturer-designed applications, allow for a wide range of useful tasks applicable to home and industrial automation.

BeeStack uses an IEEE® 802.15.4-compliant MAC/PHY layer that is not part of ZigBee itself. The PHY layer encompasses features specified by IEEE 802.15.4 for packet-based, wireless transport. The MAC sub-layer supports features specific to low-power radio frequency networks.

The NWK layer defines routing, network creation and configuration, and device synchronization. The application framework (AF) supports a rich array of services that define ZigBee functionality. ZigBee Device Objects (ZDO) implement application-level services in all nodes via profiles. A security service provider (SSP) is available to the layers that use encryption (NWK and APS).

The complete Freescale BeeStack protocol stack includes the following components:

- ZigBee Device Objects (ZDO) and ZigBee Device Profile (ZDP)
- Application Support Sub-Layer (APS)
- Application Framework (AF)
- Network (NWK) Layer
- Security Service Provider (SSP)
- IEEE 802.15.4-compliant MAC and Physical (PHY) Layers

The combined PHY, MAC, NWK, and application layer elements shown in Figure 2-1 comprise the full BeeStack implementation.

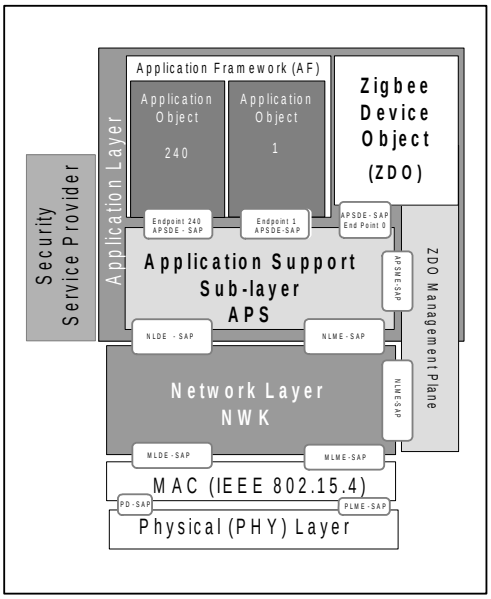

Figure 2-1. ZigBee Layers

2.1 Network Elements

A ZigBee network requires wireless devices programmed to communicate in any of several network configurations. Each network requires a device acting as a ZigBee coordinator and at least one other device with which to communicate.

2.1.1 Device Types

A ZigBee network is formed when a device declares itself a ZigBee coordinator (ZC) and permits other nodes to join its network. ZigBee routers (ZRs) and ZigBee end devices (ZEDs) can join the network either by joining the ZC directly or by joining ZRs that have already joined. ZRs permit nodes on the network to communicate with each other even if they are not within radio range because ZRs and the ZC can pass messages between nodes. ZEDs cannot pass messages between nodes; they can only send their own messages and receive messages meant for them. The ZC also serves as the trust center when the network employs security.

2.1.1.1 ZigBee Coordinator

The ZigBee coordinator roles include:

- Starting a network
- Selecting a Personal Area Network Identifier (PAN ID) for the network
- Allowing devices to join or leave the network
- Performing all the functions of a ZigBee router
- Containing the trust center in a secure network

2.1.1.2 ZigBee Router

The ZigBee router serves to:

- Route data between ZigBee devices
- Allow devices to join or leave the network
- Manage messages for its children that are end devices
- Optionally perform all the functions of a ZigBee end device

2.1.1.3 ZigBee End Device

The ZigBee end device is a reduced-function device that can:

- Sleep to save power, so it could be battery powered
- Require fewer memory resources because it does not store network-wide information or need to be able to perform network-related services

ZEDs perform functions such as switching a light on or off or monitoring an occupancy sensor. If the ZED primarily reports a sensor's state, it may sleep between measurements. For a ZED reporting the state of a switch, it can sleep until the switch is pressed, which might not occur for years. For the simplest end devices, a common design goal is to have the node run on primary batteries for the length of the batteries' shelf life.

2.1.1.4 Nodes

The collection of independent device descriptions and applications residing in a single unit, and sharing a common 802.15.4 radio, defines a *node* in a ZigBee network. Theoretically, a ZigBee network can handle more than 65,000 nodes.

Three network types common to ZigBee include the star, tree, and mesh configurations. Each network must have one coordinator, and it will have at least one other device.

ZigBee networks employ a parent and child structure. A network forms when a device declares itself a ZC and permits other nodes to join. The nodes joining that ZC become its children, and the ZC their parent. Network parents have specific responsibilities. In some network types, ZEDs communicate only with their parents, and that parent routes the ZED's messages to another destination when required.

2.1.2 Star Network

A conventional star network consists of a coordinator with one or more ZEDs associated directly with the ZC. In the star network shown in Figure 2-2, all other devices directly communicate with the ZigBee Coordinator, and the coordinator passes all messages between end devices.

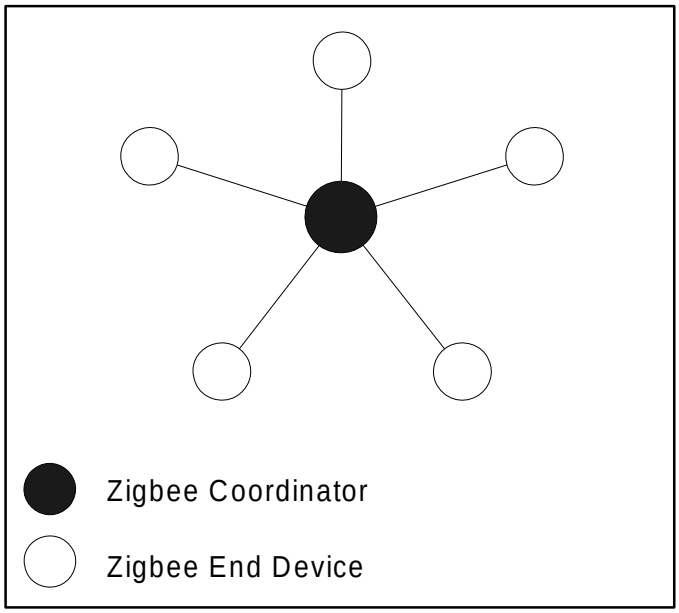

Figure 2-2. Standard Star Network Configuration

2.1.3 Tree Network

As shown in Figure 2-3, a tree network consists of a ZC with one or more routers and, optionally, one or more ZEDs associated in a hierarchical structure. A tree network extends the star network with the use of ZigBee routers (ZR).

All messages in a tree network move up or down the parent-child hierarchy. Each transfer from one node to the next is a *hop*. The depth of a tree network is the maximum number of hops a message must make to get from a source to a destination.

Every router can examine a message it has received to tell if the recipient is below it in the tree. If the recipient is not one below it, the router will pass the message to its own parent.

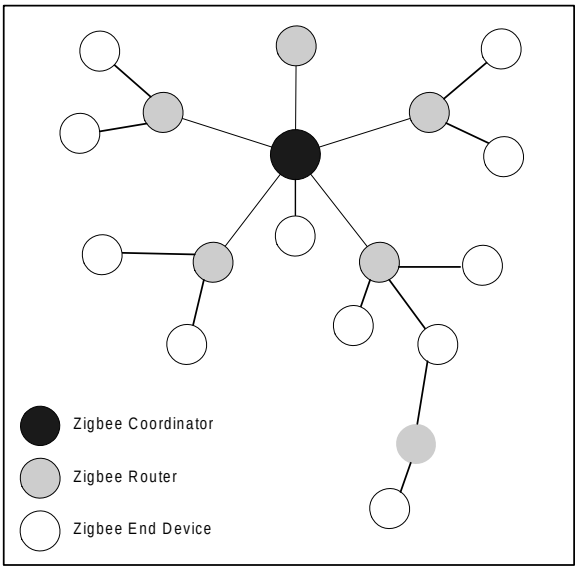

Figure 2-3. Typical Tree Network

2.1.4 Mesh Network

In a mesh network, each device can communicate directly with other devices in the network. A mesh network consists of a ZC that has one or more ZRs and optionally one or more associated ZEDs.

Figure 2-3 shows a simple ZigBee mesh network. Any device in a mesh network may send a message addressed to any other device in the network. If the two devices are within radio range of each other, the message moves in one hop, and no other devices are involved. If they are beyond each other's radio range, the message must travel from router to router, following a path that the network establishes based on its routing efficiency.

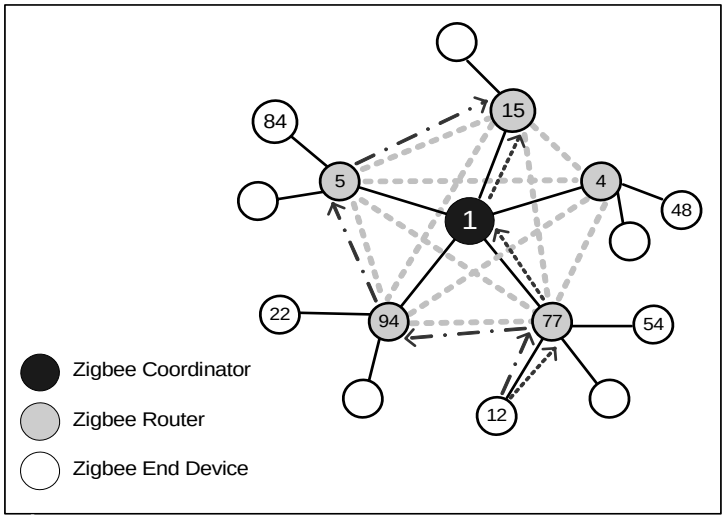

Figure 2-4. Mesh Network Configuration

2.1.5 Personal Area Network

The personal area network (PAN) encompasses a unique address space on a radio channel. A PAN resides on a channel, and the same PAN identifier may be used by another network in radio range without conflict only on a different channel. In the future, channel hopping may be permitted; for now, a PAN forms on one channel only.

NOTE

A ZigBee coordinator starts the network; however, the ZC is not required for the network to continue to function. This means that in the event of a ZC failure, it will not necessarily take down the entire network.

Networks can be extended by defining a device as a router (ZR), in a role similar to that of conventional network routers. The ZR manages routing and provides access to its child devices. A ZR can act as a ZED, and a ZC can play the role of a ZR, establishing communication paths and managing network traffic.

Networks can be structured in conventional star, tree, or mesh-tree configurations, as required by the user's application.

2.1.6 Channels

Channels are defined in accordance with the IEEE 802.15.4 specification. BeeStack applications use channels 11-26 in the 2.4 GHz range. For more detail on channel assignments, see Chapter 8, "Network Layer".

2.1.7 Device and Service Discovery

ZigBee devices discover other ZigBee devices by broadcasting or unicasting a message. Devices send one of two forms of device discovery requests, an IEEE address request and a NWK address request. Service discovery allows a node to find nodes that offer services it needs or nodes that need services it offers. For more information, see Chapter 7, "ZigBee Device Profile".

2.1.8 Addressing/Messaging

Messages can be sent from one device to another once devices have identified each other. The commands sent to application objects at the destination address include the node's address and the source and destination endpoint.

There are five addressing modes in ZigBee:

- 16-bit direct: short, or network, address
- 64-bit direct: long, or IEEE address
- Indirect (uses local binding table)
- Broadcast
  - All nodes
  - All routers and the coordinator
  - All nodes that constantly monitor network traffic (RxOnWhenIdle)

  - All endpoints on an individual node
- Group

Direct addressing requires the sending device to know the target device's attributes:

- Address (which node?)
- Endpoint (which application within the node?)
- Cluster identifier (which object within the application?)

Every IEEE 802.15.4 radio has a 64-bit address that is unique in the world. Every node in a ZigBee network has a 16-bit network address that is unique within that network. ZigBee does not send messages with 64-bit addresses. When a ZigBee application tells the stack to send a message to an IEEE address, the stack must find out what network address that node has before sending the message with the network address.

Indirect addressing mode uses a local binding table to determine the destination node(s). The local binding table can hold multiple destinations (destinations are always either direct-64 or group destinations). A single data request can end up at multiple destinations, depending on the binding information. The binding between source and destination must be established before the source node can use indirect addressing. Every entry in the local binding table that contains the same source address as the data request are considered destinations.

Nodes can broadcast messages in several ways. An application can send a message to an individual network address and the endpoint 0xFF, which will cause the receiving stack to deliver that message to every endpoint on the node. A message with 0xFFFF as the destination address goes to every node on the network (within the specified radius of the message). A message to 0xFFFD goes to every device that is always on (RxOnWhenIdle = TRUE). A message to 0xFFFC goes to every router and the coordinator.

Messages addressed to specific endpoints on a collection of devices use a single group address. That group address may then be used to direct outgoing clusters, as well as the attributes contained in them, to each of the devices and endpoints assigned to the group. Group addressing uses a 16-bit destination address with the group address flag set in the APS frame control field. Included in the source are the cluster identifier, profile identifier and source endpoint fields in the APS frame.

Endpoints require a form of sub-addressing in conjunction with the mechanisms of IEEE 802.15.4. An endpoint number identifies individual switches and lamps, for example. A switch might use endpoint 5, while a second switch might use endpoint 12.Each lamp that these switches control has its own endpoint number. Endpoint 0 is reserved for device management. Each identifiable sub-unit in a node (such as the switches and lamps) has its own specific endpoint address in the range 1-240.

2.1.9 Binding

Binding creates logical links between endpoints on devices, allowing them to work together to perform specific tasks. Binding maintains information on each logical link in a binding table. The ZC or the source device of the binding pair maintains the binding table for the network.

As shown in Figure 2-5, binding creates relationships between applications. For example, a single network may contain many lights and switches, and binding allows any switch to control either a particular light or a group of lights.

NOTE

Binding is unidirectional; a switch binds to a light, but not the light to the switch.

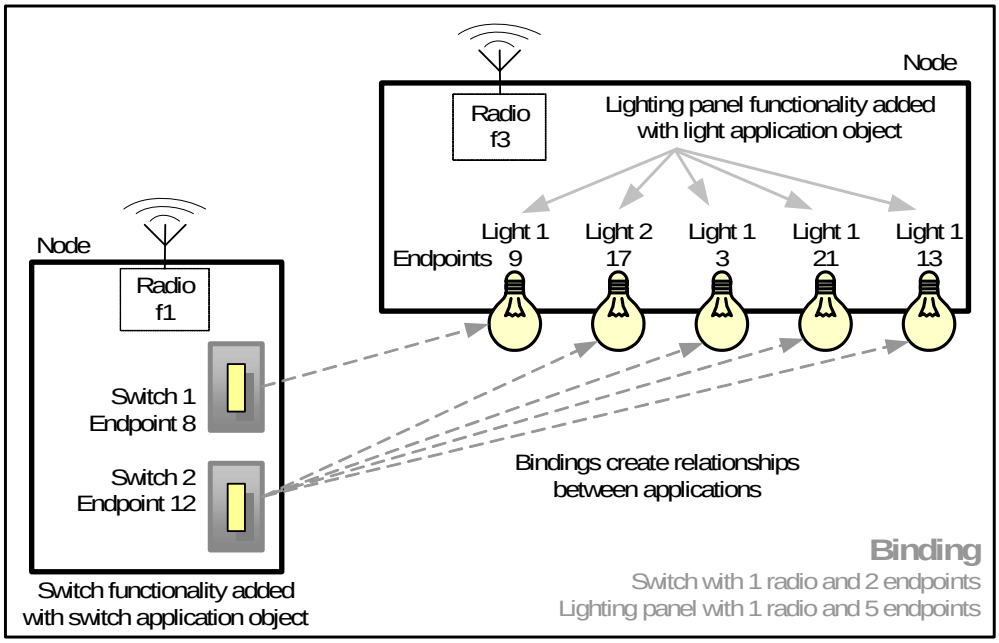

Figure 2-5. Binding and Application Objects

2.2 Application Elements

This section introduces the application concepts, which are then detailed in later chapters along with code examples, to help designers and developers in creating new BeeStack applications.

2.2.1 Applications

Application objects define the activities and functions in BeeStack. Each application runs as a component of the top portion of the application layer. The manufacturers that implement the various applications define both the applications and their functionalities.

Broad areas of applications, such as building automation or home automation, fall into specific application domains.

Alternatively, an application profile may create sub-types within the cluster known as attributes. In this case, the cluster is a collection of attributes specified to accompany a specific cluster identifier. Binding decisions are made by matching the output cluster identifier to an input cluster identifier, assuming both exist in the same profile.

Every application in BeeKit starts with `BeekitAppInit.c`. See Chapter 3, “BeeStack Features” for more information.

2.2.2 Attributes

In BeeStack, an attribute is a data entity representing a physical quantity or state, a data item to read or write. Data is communicated to other devices using commands with attributes included.

For example, a wireless UART has only clusters, and no attributes, while an ON/OFF light application uses both the ZigBee cluster library (ZCL) and a home automation profile.

2.2.3 Clusters

Clusters contain the data flowing into or out of a device. The 16-bit cluster identifier, which is unique within the application segment, identifies a specific cluster. Clusters can be thought of as behaving the same way a port might in a traditional network. Within the protocol stack, the message sent from a client gets directed to a specific point on the server side, and the attributes direct that message to the correct port, or cluster.

The ZigBee device profile (ZDP) sends commands and responses contained in clusters, with the cluster identifiers enumerated for each command and response. Each ZDP message is then defined as a cluster.

For example, an ON/OFF cluster sends a command from the client (the switch) to turn on or off an entity on the server (the light). ZCL acts as a repository for cluster functionality. The ON/OFF message defines one single attribute, containing the device's status in binary form.

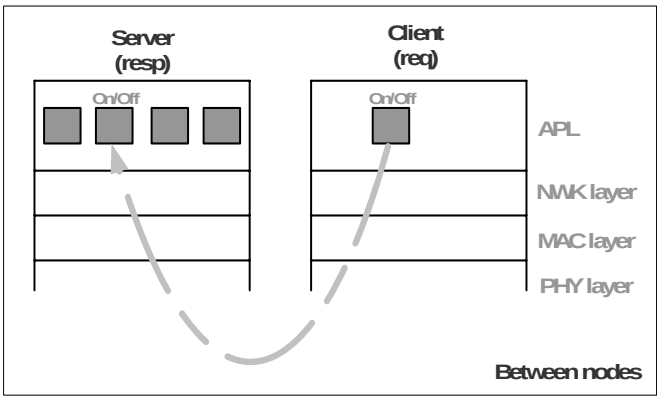

Figure 2-6. On and Off Lighting Application Stack Behavior

An application or profile uses the ZigBee cluster library (ZCL) to complete its work.

2.2.4 Endpoints

Applications reside on endpoints, which act as independent objects. The number assigned to an endpoint is essentially the application's address within the ZigBee device. This allows other devices to communicate separately with each application on a device.

BeeStack provides services to allow endpoints to find other endpoints on the network with which they can communicate to perform their intended tasks. An application can send a message to all endpoints using gZbBroadcastEndPoint_c.

```
#define gZbBroadcastEndPoint_c 0xff
```

A single device can have as many as 240 user application endpoints, and each endpoint can be independent of the others. The ZDO resides as a separate application on endpoint 0.

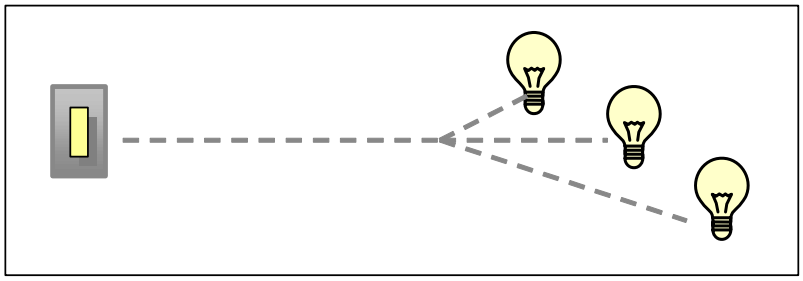

Figure 2-7. Endpoints in ZigBee Network

Endpoints play three major roles in BeeStack, allowing:

- Different application profiles to exist within each node
- Separate control points to exist within each node
- Separate sensors or other devices to exist within each node

Chapter 3
BeeStack Features

BeeStack initializes itself by doing the following:

- Initializes the MAC and PHY layers
- Initializes the Timer Module
- Initializes the serial ports
- Switches off all the LEDs on the board
- Initializes the APS Layer
- Initializes the Application Framework
- Initializes the ZigBee Device Objects
- Initializes the NWK Layer
- Initializes the NVM Module

Every application starts in BeeAppInit(). The application task functions are called during initialization, along with any application-specific initializations. Those commands include, for example, hardware initialization and set up, table initialization, and power-up notification.

The function BeeStackInit() can be found in the `BeeStackInit.c` file; its initialization API is:

```
void BeeAppInit(void);
```

3.1 BeeStack Task Scheduler

BeeStack uses co-operative multi-tasking. Each task is a separate function that must relinquish control often enough for the BeeStack components to get their work done in a timely manner. The BeeStack task scheduler runs when the running task releases control. The tasks have fixed priorities, and the task scheduler starts the highest-priority task that has an event waiting for it. If there are no tasks with events waiting, the scheduler runs the idle task. The idle task has work of its own to do. The task scheduler is provided as source code.

The task scheduler must be configured before it can be used to run BeeStack and the application as tasks. Users configure Task Scheduler in BeeKit, defining the number of tasks that it handles, the task entry point for each of the tasks, the priority of the tasks, and other configuration parameters.

The `TS_Interface.h` file configures Task Scheduler. The file is found in the following path:

```
<Installation Folder>\BeeStack\SSM\TS\Interface
```

A global task list defines the set of initial tasks in the application space, including at least one application task. Optionally, tasks can be created or destroyed at run-time.

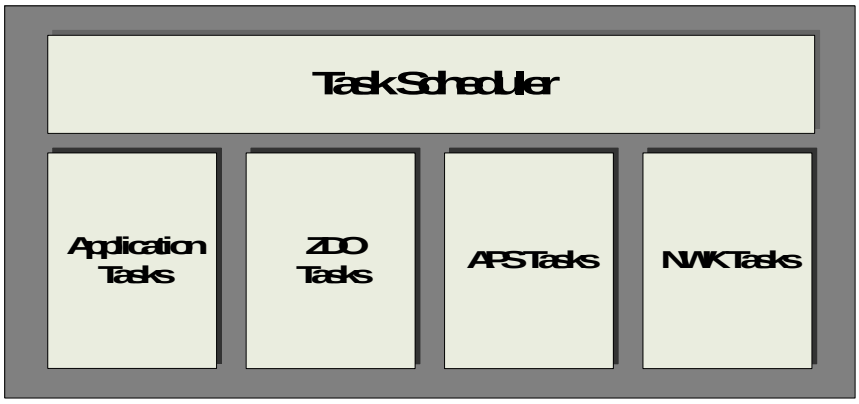

Figure 3-1. Task Scheduler Functionality

This macro defines the task scheduler interface:

```
#define _TS_INTERFACE_H_
```

For more information on the Task Scheduler, refer to the *Freescale BeeStack Platform Reference Manual*, (FSPRM)

3.2 BeeStack Application Programming Interface

This reference manual explains the BeeStack function calls and application programming interfaces (APIs). The functions fall into two categories: synchronous and asynchronous calls.

Synchronous calls return an immediate response, in some cases with an error code. Examples include the functional calls AF_MsgAlloc() and NlmeGetRequest().

Asynchronous calls start a process that may take seconds to complete; for example, AF_DataRequest() sends a packet over the air to another node in the ZigBee network. Asynchronous calls have a callback "confirm" function.

Users may customize BeeStack using the parameters and options in the files listed in Table 3-1.

Table 3-1. BeeStack User-Configurable Files

File Name	Description
`ApplicationConf.h`	Contains the main configuration values (PAN ID, Channel)
`BeeStackConfiguration.h`	Sets BeeStack table sizes and some compile time BeeStack parameters

For more information on the user-configurable options in BeeStack, refer to Chapter 11, "User-Configurable BeeStack Options".

NOTE

The tables that follow are informational only. BeeKit sets all the configurable parameters, and no further user intervention is required.

Table 3-2 lists files that describe the BeeStack APIs. BeeStack includes both mandatory and optional files.

Table 3-2. BeeStack APIs

Include File Name	Description
`ApplicationConf.h`	General application configuration options (PAN ID, channel)
`BeeStackConfiguration.h`	ZDP and stack level configuration options
`AppToAfInterface.h`	Prototypes for AF layer calls, including sending and receiving messages over the air
`ASL_ZdpInterface.h`	Prototypes and types for interacting with ZDP (over-the-air) API
`ASL_UserInterface.h`	Prototypes and types for interacting with common app UI (App Support Layer) API
`BeeStackInterface.h`	Prototypes for interacting with information bases (AIB, NIB)
`BeeAppInit.h`	Minimal application API (for use without ASL UI)

The BeeStack source files listed in Table 3-3 must be included in the application project workspace.

Table 3-3. Required BeeStack Source Files

Function	Description
`AppStackImpl.c`	This source file implements the Channel and PAN ID selection logic.
`BeeStackInit.c`	This file implements BeeStack initialization.
`BeeStackUtil.c`	This file contains the implementation of the functions used to save and restore information from the non volatile memory.
`ZbAppInterface.c`	This file contains the functions and declarations used to register the endpoints 0 and 255 for both compile time and run time registration. It is also used to register the application endpoints at compile time.

3.3 Source Files – Directory Structure

BeeStack files use the following extensions:

- Source code: `.c`, `.h`
- Libraries: `.lib`
- S19 record format targets: `.s19`
- Memory maps: `.map`

Figure 3-2 shows the directory structure used for the application libraries and source files.

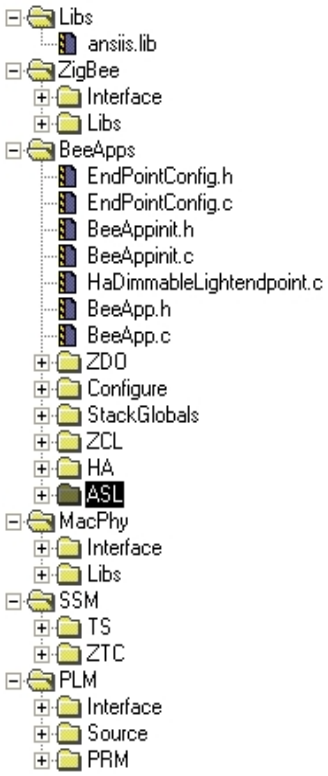

Figure 3-2. Library and Application File Directories

3.4 Miscellaneous Source Files

This section describes the source files in BeeStack which are not already described elsewhere.

Table 3-4. Required BeeStack Source Files

File	Description
BeeStack_Globals.c	Contains the globals that interface with the other layers (APS, ZDO, NWK). For example, this file defines the size of the routing table and contains the routing table array.
BeeStack_Globals.h	This file contains prototypes and types for BeeStack_Globals.c.
BeeStackParameters.h	This file contains the type beeStackParameters_t, which contains binding and security information.
CSkipCalc.c	This file contains Macros that will calculate CSkip values for a given set of max_children, max_depth and max_routers. It does not error check, so the values must be in range to result in less than 0xfff0 nodes in the network.
ZbAppInterface.c	Contains endpoint 0 description used by ZDP.
ZbAppInterface.h	Header for ZbAppInterface.c.

Chapter 4
Application Framework

The BeeStack application framework (AF) defines the environment in which ZigBee devices host application objects. Inside the application framework, application objects send and receive data through the service access point (SAP) handlers. For example, the application sub-layer data entity service access point (APSDE-SAP) controls data services between the application objects and the APS layer.

Layers in BeeStack communicate with each other by passing messages through SAP handler functions. Communication with a next higher or lower layer involves two SAP handler functions. Effectively, one SAP handler deals with messages from a layer to its next higher layer, and a second SAP handler manages the messages from the next higher layer back to the lower layer.

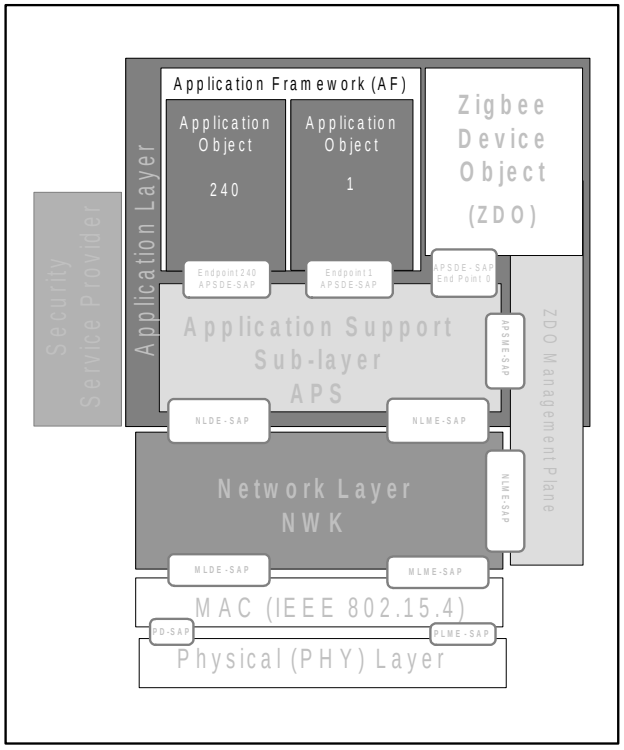

Figure 4-1. BeeStack Layers and Application Framework

The APSDE-SAP data services include the request, confirm, response and indication primitives for data transfer.

- A request primitive transfers information between peer application object entities
- A confirm primitive reports the results of a request function call
- A response primitive returns errors, acknowledgements, or other information

- An indication primitive communicates the transfer of data from the APS to the destination application object entity

Up to 240 distinct application objects can be defined, with each interface on an endpoint indexed from 1 to 240. ZigBee defines two additional endpoints for APSDE-SAP use:

- Endpoint 0 for the data interface to the ZDO
- Endpoint 255 for the data interface function to broadcast data to all application objects

Endpoints 241-254 are reserved for future use.

Through the ZigBee device objects (ZDO) public interfaces, the application objects provide:

- Control and management of the protocol layers in a ZigBee device
- Initiation of standard network functions

In BeeStack, applications never call on SAP handlers directly, but instead use a set of service functions. The AF service functions are described below.

4.1 AF Types

BeeStack AF types described in Table 4-1 constitute a partial list.

Table 4-1. Application Framework Types and Definitions

Type	Description
afAddrInfo_t	Provides complete address information for AF_DataRequest()
afDefaultRadius_c	Defines default number of hops a message takes to reach a destination address
afDeviceDef_t	Defines device type
zbEndPoint_t	Defines an endpoint
zbStatus_t	Return value (status) of a function or service
zbIeeeAddr_t	Long address. Also used for extended PAN ID.
zbPanId_t	Identifies a single ZigBee network (part of IEEE 802.15.4)
zbNwkAddr_t	Short address. All ZigBee data packets use the short address.
zbClusterId_t	APS-level data. Determines both ZDO and application commands.
zbGroupId_t	APS-level group addressing
zbProfileId_t	Cluster IDs defined within a profile
zbDeviceId_t	Device IDs defined within a profile
zbSceneId_t	Defines the cluster ID for a programmed scheme (scene)
zbAddrMode_t	Defines addressing mode

4.2 Endpoint Management

An application must register its endpoint with BeeStack before it can communicate with other devices. Applications on the endpoints of other devices on the network find objects to communicate with using this information. BeeStack application profiles use endpoints as application identifiers within a ZigBee device.

4.2.1 Simple Descriptor

The simple descriptor contains the description of an endpoint. Every endPointDesc_t structure points to a simple descriptor structure.

The simple descriptor structure provides information to BeeStack about an endpoint. BeeStack uses this declaration syntax for the zbSimpleDescriptor_t, defined in `BeeStack_Globals.h`:

```
typedef struct zbZbSimpleDescriptor_tag
{
    /*End point ID */
    zbEndPoint_t  endPoint;
    /*Application Profile ID*/
    zbProfileId_t aAppProfId;
    /*Appliacation Device ID*/
    zbDeviceId_t  aAppDeviceId;
    /*Application Device Version And APS Flag*/
    uint8_t       appDevVerAndFlag;
    /*Number of Input Cluster ID Supported by the End Point*/
    zbCounter_t   appNumInClusters;
    /*Place Holder for the list of Input Cluster ID*/
    uint8_t      *pAppInClusterList;
    /*Number of Output Cluster ID Supported by the End Point*/
    zbCounter_t   appNumOutClusters;
    /*Place Holder for the list of Output Cluster ID*/
    uint8_t      *pAppOutClusterList;
}zbZbSimpleDescriptor_t;
```

4.2.2 Register Endpoint

Endpoints must register on the network before they can communicate with other devices. AF_RegisterEndPoint() allows the application to receive data indications and confirms.

BeeStack endpoint registration uses two function types, the endpoint descriptor, `endPointDesc_t`, and the simple descriptor, zbSimpleDescriptor_t.

This is generally accomplished in the BeeAppInit() function.

Prototype

```
zbStatus_t AF_RegisterEndPoint(const endPointDesc_t * pEndPoint);
```

4.2.3 De-register Endpoint

An endpoint de-registers, or removes itself from the network, with the function AF_DeRegisterEndPoint().

Prototype

```
zbStatus_t AF_DeRegisterEndPoint(zbEndPoint_t endPoint);
```

4.2.4 Get Endpoint

An endpoint is an ID number (1-240) that refers to a single object or widget in a node. Using the endpoint, the ZigBee defined Simple Descriptor can be retrieved (see section 4.2.5) which describes the input and output clusters and other parameters.

In addition the BeeStack defined Device Definition can be retrieved, which contains ZigBee Cluster Library information, including the instantiation of an endpoint's data. If a node contained two OnOffLights, then each light could be controlled independently on two separate endpoints, and the Device Definition for that endpoint would refer to those two sets of data.

The function AF_GetEndPointDevice() retries the Device Definition from an endpoint.

Prototype

```
afDeviceDef_t *AF_GetEndPointDevice(zbEndPoint_t endPoint);
```

4.2.5 Find Endpoint Descriptor

`AF_FindEndPointDescriptor()` allows an application (or the stack) to convert from an endpoint number (1-240) to a SimpleDescriptor, as described by the ZigBee specification. This helper function looks up information contained in the simple descriptor, such as the application profile ID, application version, or in and out clusters. See the definition of Simple Descriptor for more details.

This function `AF_FindEndPointDescriptor` searches for the endpoint simple descriptor based on the endpoint ID (0x00-0xf0).

Prototype

```
zbSimpleDescriptor_t* AF_FindEndPointDesc(uint8_t endPoint);
```

Returns

- Pointer to the simple descriptor
- NULL, if not a registered endpoint

4.3 Message Allocation

`AF_MsgAlloc()` allows an application to allocate a message for building a larger packet to be sent using `AF_DataRequestNoCopy()`. Normally, the command `AF_DataRequest()`only copies the payload. The use of the `AF_MsgAlloc()` and `AF_DataRequestNoCopy()` allows an over-the-air message to be built in-place, saving some RAM.

BeeStack uses a pool of messages to prevent heap fragmentation used on some other ZigBee implementation. Each message is of fixed size in an array. Over-the-air messages are called "big buffers". There is a limited number of these buffers available (generally 5 or 6, depending on node type). See gTotalBigMsgs_d in AppToMacPhyConfig.h to set the number of big buffers. The Freescale MAC documentation contains more details on the message buffer system.

Prototype

```
void * AF_MsgAlloc(uint8_t payloadLen);
```

Returns

- MSG_Alloc(gMaxRxTxDataLength_c);

4.4 AF Data Requests

Application Framework (AF) data requests are the primary way applications send data over the air to one or more other nodes in a ZigBee network.

There are two functions for sending data, one that copies the application’s data payload, AF_DataRequest(), and one that leaves the payload in place, AF_DataRequestNoCopy().

Data requests are asynchronous calls. They may take several seconds to process if they need to wait for a response from another node in the network, allow for retries, or make multiple hops. For example, if using APS ACKs (application-level end-to-end acknowledgement), it will take up to 4.5 seconds to indicate failure to deliver the packet on the data confirm.

`AF_DataRequest()` sends a message to the APS layer, which sends the packet to the NWK layer, the MAC and eventually out the radio.

```
zbStatus_t AF_DataRequest(afAddrInfo_t *pAddrInfo, uint8_t payloadLen, void *pPayload,
zbApsCounter_t *pConfirmId);
```

Because the payload is copied in `AF_DataRequest()`, the payload may be a local variable on the C stack, or a global or any other data location.

For each `AF_DataRequest()`, there is exactly one data confirm. The data confirm comes back to the application in the function `BeeAppDataConfirm()`. See the file BeeApp.c in any project.

Confirms may come in a different order than they were sent, due to retries and delivery times of the packet across the network. For example, if an application sends out 2 data requests one after the other, and the first one needs to retry due to noise on the channel and the second one does not, the confirm will come in for the second `AF_DataRequest()` before the confirm for the first `AF_DataRequest()`.

The easiest (and recommended) method is for an application to only send out one `AF_DataRequest()` at a time, and wait until the confirm before sending out another data request.

Alternately, an application can keep track of the confirm ID by providing a pointer to the AF_DataRequest() in the pConfirmId parameter, to match the confirm IDs coming into `BeeAppDataConfirm()`. The pConfirmId parameter can be NULL if using the one-at-a-time method.

Sometimes an application needs to send many bytes of data as a payload on a given packet. In this case, the AF framework provides a no-copy interface for data requests. A general rule of thumb is to use the AF_DataRequestNoCopy() for payloads of more than 32 bytes, or variable length payloads that could be more than 32 bytes. The prototype for the no-copy data request is as follows:

```
zbStatus_t AF_DataRequestNoCopy(afAddrInfo_t *pAddrInfo, uint8_t payloadLen,
afToApsdeMessage_t *pMsg, zbApsCounter_t *pConfirmId);
```

Instead of the pPayload parameter there is a pMsg parameter. This message buffer is of the type used to send to SAP handlers directly. Because of this, special care must be taken, as a message buffer leak could cause the node to stop sending/receiving data (as it could run out of message buffers).

To allocate a message buffer, use the function

```
void *AF_MsgAlloc(void);
```

For example:

```
void SendMaxPacket(afAddrInfo_t *pAddrInfo)
{
        afToApsdeMessage_t *pMsg;
        uint8_t *pPayload
        uint8_t maxLen;

        pMsg = AF_MsgAlloc();
        pPayload = AF_Payload(pMsg);
        maxLen = AF_MaxPayloadLen();

        /* fill entire payload with 0x33 */
        FLib_MemSet(pPayload, 0x33, maxLen);
        AF_DataRequestNoCopy(pAddrInfo, maxLen, pMsg, NULL);
}
```

The lower layers (APS, NWK or MAC) will free the message buffer allocated for data requests.

For both `AF_DataRequest()` and `AF_DataRequestNoCopy()`, The afAddrInfo_t structure is used to define the destination of the packet. The structure is as follows:

```
typedef struct afAddrInfo_tag
{
        zbAddrMode_tdstAddrMode;/* ind, group, 16, 64 */
        zbApsAddr_tdstAddr;/* short, long or group */
        zbEndPoint_tdstEndPoint;/* destination endpoint */
        zbClusterId_taClusterId;/* cluster to send */
        zbEndPoint_tsrcEndPoint;/* source endpoint */
        zbApsTxOption_t txOptions;/* ACK */
        uint8_t  radiusCounter;/* radius */
} afAddrInfo_t;
```

Once a node is on a network, it can communicate to any other node on the network. There is no need for binding or setting up groups. All the sending node needs is the 16-bit short address of the receiving node.

The destination address mode (dstAddrMode) affects the rest of the destination fields; it may be one of the following:

- gZbAddrModeIndirect_c — ignores dstAddr and dstEndPoint because the destination is found in the local binding table based on the srcEndPoint field.
- gZbAddrModeGroup_c — ignores dstEndPoint because that is always the broadcast endpoint (0xff) on groups. dstAddr is a 16-bit group address.
- gZbAddrMode16Bit_c — uses both dstEndPoint and a 16-bit dstAddr.
- gZbAddrMode64Bit_c — uses dstEndPoint and a 64-bit dstAddr. Note that the 64-bit address is converted locally to a 16-bit address before being sent out the radio. ZigBee always uses 16-bit addresses, even though IEEE 802.15.4 can use 16-bit or 64-bit addresses in its messages. Make sure to call ASL_NWK_addr_req() for the destination node before using 64-bit address mode.

The local binding table is set up through local or remote binding commands. Local binding commands use APS functions such as APSME_BindRequest(). Remote binding commands use ZDP functions such as ASL_EndDeviceBindRequest(). Groups are set up locally in a node using APS functions such as APSME_AddGroupRequest() or remotely in other nodes using ZigBee Cluster Library (ZCL) functions.

The 16-bit destination address may be the address of the node or one of the following broadcast addresses:

- gaBroadcastAddress – broadcast to all nodes
- gaBroadcastZCnZR – broadcast only to routers (no end devices)
- gaBroadcastRxOnIdle – broadcast to all constantly-awake (RxOnIdle) devices

The cluster ID is up to the application. BeeStack puts no restrictions on clusters. There is a structure in EndPointConfig.c called zbSimpleDescriptor_t. This structure, the simple descriptor, is used for over-the-air discovery of services, but it is not used for cluster filtering.

The source endpoint must be a registered endpoint. See AF_RegisterEndPoint() and the BeeAppInit() function.

The txOptions allow a few transmit options including

- gApsTxOptionNone_c — use no TxOptions.
- gApsTxOptionSecEnabled_c — Enable security on this packet (requires security to be selected in BeeKit).
- gApsTxOptionAckTx_c — Enable acknowledgements and reliable transmission. By default, the data confirm indicates the data was sent. With ACK turned on, the data confirm indicates whether the receiving node received the packet. ACKs cause more network traffic.
- gApsTxOptionSuppressRouteDiscovery_c — Normally, packets discover a route if needed. Turn off route discovery to route along the tree.
- gApsTxOptionForceRouteDiscovery_c — Normally, packets discover a route if needed. Turn on force route discovery to discover a route before sending the packet.

The radius field tells ZigBee how far to send the packet before expiring the packet. Set this parameter to 0 to use the default of afDefaultRadius_c, which is twice network depth, or 10 in the Home Controls Stack Profile 0x01.

4.5 AF Data Indications

When an `AF_DataRequest()` sent by one node is received by another node, the results come into the receiving node in the function `BeeAppDataIndication()` in the file BeeApp.c. The function typically looks as follows:

```
void BeeAppDataIndication(void)
{
  apsdeToAfMessage_t *pMsg;
  zbApsdeDataIndication_t *pIndication;
  zbStatus_t status = gZclMfgSpecific_c;

  while(MSG_Pending(&gAppDataIndicationQueue))
  {
    /* Get a message from a queue */
    pMsg = MSG_DeQueue( &gAppDataIndicationQueue );

    /* give ZCL first crack at the frame */
    pIndication = &(pMsg->msgData.dataIndication);
    status = ZCL_InterpretFrame(pIndication);

    /* not handled by ZCL interface ... */
    if(status == gZclMfgSpecific_c)
    {
      /* insert manufacturer specific code here... */
    }

    /* Free memory allocated by data indication */
    MSG_Free(pMsg);
  }
}
```

NOTE

It is up to the application to free the message buffer. BeeStack is designed this way so the application can keep the message for awhile if it needs to do further processing on the message that may take time and the application wishes to relinquish control to other tasks meanwhile. Be very careful to free message buffers. If the message buffers are not freed, the system may run out, which would prevent further ZigBee communication.

In the above example, the ZigBee Cluster Library (ZCL) is used to interpret the frame, possibly getting or setting attributes, etc. Private profiles that do not use ZCL can interpret the pIndication directly, which contains all the information the application needs to understand the incoming frame.

The pIndication structure is as follows:

```
typedef struct zbApsdeDataIndication_tag
{
        zbAddrMode_tdstAddrMode;    /* address mode */
        zbNwkAddr_taDstAddr;       /* dest addr or group */
```

```
        zbEndPoint_tdstEndPoint;    /* dest endpoint */
        zbAddrMode_tsrcAddrMode;    /* always 16-bit */
        zbNwkAddr_taSrcAddr;       /* src addr or group */
        zbEndPoint_tsrcEndPoint;    /* source endpoint */
        zbProfileId_taProfileId;     /* profile ID */
        zbClusterId_taClusterId;     /* cluster ID */
        uint8_t  asduLength;     /* length of payload */
        uint8_t  *pAsdu;         /* pointer to payload */
        bool_t            fWasBroadcast;  /* was broadcast? */
        zbApsSecurityStatus_t fSecurityStatus;  /* secured? */
        uint8_t  linkQuality;    /* link quality */
} zbApsdeDataIndication_t;
```

An application can tell whether the packet was broadcast or sent directly, whether the packet was secured or not, what the link quality was on that particular packet and whether the packet was sent to a group or unicast to this individual node.

The lower layers will have already filtered packets that do not match the node criteria, do not match the profile ID of the receiving endpoint, and do not match the group (if any) on that endpoint. The lower layers also filter out duplicates so the application does not need any logic to handle duplicates packets.

The application is responsible for filtering based on clusters.

Make sure that the each endpoint to receive data is registered, using AF_RegisterEndpoint(). This is generally done in the BeeAppInit() function.

There is no attribute in the data indication. Attributes are a ZigBee Cluster Library concept and are not used in private profiles.

Chapter 5
Application Support Sub-layer

The application support sub-layer (APS) provides the interface between the NWK layer and the application layer.

The BeeStack application support sub-layer (APS) roles include:

- Maintaining tables for binding, or matching two devices together based on their services and their needs
- Forwarding messages between bound devices
- Group address definition, removal and filtering of group addressed messages
- Mapping between 64-bit IEEE addresses and 16-bit NWK addresses
- Reliable data transport

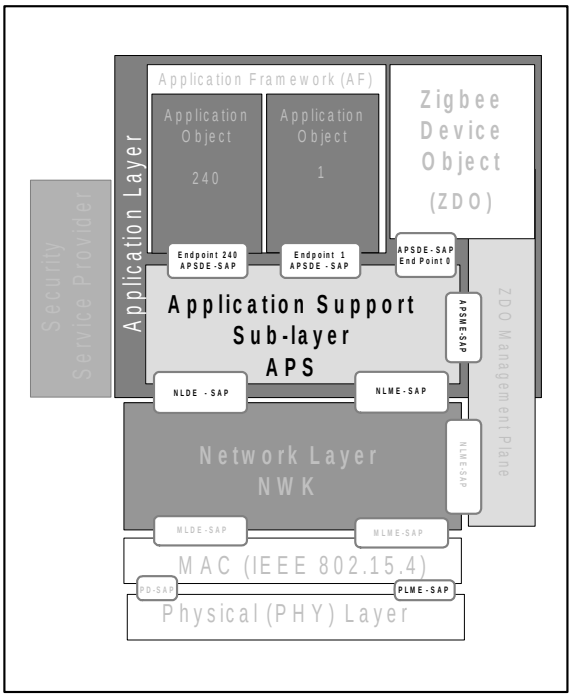

Figure 5-1. BeeStack Application Support Sub-Layer Elements

A general set of services supports communication with ZigBee device objects (ZDO) and the manufacturer-defined application objects. The APS interface to the next higher and next lower layers utilizes two entities: the data (service) entity and the management (service) entity.

- The APS data entity (APSDE) provides over-the-air data transmission service via its service access point (SAP), the APSDE-SAP

- The APS management entity (APSME) provides management service with its APSME-SAP and maintains a database of managed objects known as the APS information base (AIB)

5.1 Direct and Indirect Data Addressing

Direct addressing requires either the short or extended (also called long, MAC or IEEE) address of the target or destination device. The APS layer maintains the short address and its corresponding extended address in the AIB.

While network addresses depend on the network topology and the device's network association, the extended address is unique to the device, and does not change with the network topology.

With indirect addressing, a device sends the data without a destination address, which must be looked up in a binding table. The binding table can be on the device generating the message, or it can be on the ZC. If it is the latter, the message must go to the ZC for destination lookup and retransmission to the destination device or devices.

5.2 APS Layer Interface

In the APS layer, the APSME primitives affect the local node only, and they use internal (not ZigBee) formats for many parameters.

Use the ZDP versions of these functions when communicating to other nodes, or when using ZigBee standard over-the-air formats. The ASPDE functions affect the over-the-air (OTA) data.

The macros in this section use the given attributeId to call the appropriate macro.

5.2.1 Get Request

This macro retrieves values (attributes) from the APS information base (AIB).

```
#define ApsmeGetRequest(attributeId, pValue) \ ApsmeGetRequest_##attributeId(pValue)
```

Declaration syntax:

```
typedef struct apsmeGetReq_tag{
         uint8_t aibAttribute;
         uint8_t *pAibAttributeValue;
}apsmeGetReq_t;
```

The following attributes may be retrieved using ApsmeGetRequest():

gApsTrustCenterAddress_c	Usually 0x0000 (ZC)
gApsSecurityTimeOutPeriod_c	Timeout for authentication

The following Attributes may be retrieved using ApsmeGetRequestTableEntry():

gApsAddressMap_c	Retrieve an entry from the address map
gApsBindingTable_c	Retrieve an entry from the binding table
gApsGroupTable_c	Retrieve an entry from the group table

The ApsmeGetRequestTableEntry() macro is used when the entry is not a single item, but an array of items. It is prototyped as follows:

```
void * ApsmeGetTableEntry(uint8_t attributeId, uint8_t index);
```

The return is one item in the table. Care must be taken not to read past the end of the table. The index should be 0 - (n-1), where n is the number of elements in the table. Use gMaxAddressMapEntries, gMaxBindingEntries, and giMaxGroups respectively. Note that the tables are in BeeStack internal form. Use the ZDP Mgmt functions to retrieve the entries in ZigBee over-the-air form. The types for the returned entry are:

- apsBindingTable_t
- zbAddressMap_t
- zbGroupTable_t

5.2.2 Set Request

ZDO uses this macro to set a simple attribute in the AIB.

```
#define ApsmeSetRequest(attributeId, pValue)
ApsmeSetRequest_##attributeId(pValue
```

Use ApsmeGetRequestTableEntry() for table entries. This function is prototyped as:

```
void ApsmeSetRequestTableEntry
(
  uint8_t attributeId,
  uint8_t index,
  void *pValue
);
```

5.2.3 Get Table Entry

This macro requests an entry from an AIB table attribute (for example, an address map).

```
#define ApsmeGetRequestTableEntry(attributeId,index) \ ApsmeGetRequest_##attributeId(index)
```

5.2.4 Set Table Entry

This macro attempts to set an entry in an AIB table.

```
#define ApsmeSetRequestTableEntry(attributeId,index,pValue) \
ApsmeSetRequest_##attributeId(index,pValue)
```

5.2.5 Add to Address Map

This function, `APS_AddToAddressMap` , seeks to add a device's IEEE address to the network address table.

```
addrMapIndex_t APS_AddToAddressMap(zbIeeeAddr_t aExtAddr, zbNwkAddr_t aNwkAddr);
```

The index returned is either 0 - (gMaxAddressMapEntries-1), or gAddressMapFull_c to indicate it couldn't be added because the table is full. The address map associates a 16-bit NWK address with a 64-bit IEEE address, and is updated automatically with end-device-anounce.

5.2.6 Remove from Address Map

The function APS_RemoveFromAddressMap() removes from the address map an entry found by its IEEE address. It does not return a status, and will do nothing if the address is not already in the address map. The only parameter is an IEEE address to remove.

Prototype

```
void APS_RemoveFromAddressMap(zbIeeeAddr_t aExtAddr);
```

5.2.7 Find IEEE Address in Address Map

The function `APS_FindIeeeInAddressMap()` initiates a search through the address map for the IEEE address.

Prototype

```
addrMapIndex_t APS_FindIeeeInAddressMap(zbIeeeAddr_t aExtAddr);Returns
```

Returns

- Index into that item, if found
- gNotInAddressMap_c, if not found

5.2.8 Get NWK Address from IEEE Address

The function `APS_GetNwkAddress` requests the network address based on a device's IEEE address.

Prototype

```
uint8_t* APS_GetNwkAddress( uint8_t * pExtAddr);
```

Returns

- Pointer to the NWK (short) address, given an IEEE (long) address
- NULL, if not self or in address map

5.2.9 Get IEEE Address from NWK Address

The function `APS_GetIeeeAddress` requests the IEEE address based on known NWK address.

Prototype

```
uint8_t *APS_GetIeeeAddress(uint8_t *pNwkAddr);
```

Returns

- Pointer to the IEEE (long) address, given a NWK (short) address
- NULL, if not self or in address map

5.3 Binding

Binding creates logical links between application devices and endpoints to allow them to work together to perform specific tasks. Binding maintains information on each logical link in a binding table. Each source node maintains its own binding table, or the ZC maintains the binding table for the network.

Binding is not necessary for ZigBee communication. Group or direct mode can be used instead. However, binding can be useful because binding tables are automatically updated if an end-device node moves to a new parent in the network. Binding also allows the destination address/endpoint information to be set up by a commissioning tool.

ZDO issues a primitive to the APS layer to initiate the binding operation on a device that supports a binding table. This in-memory association has no over-the-air behavior.

5.3.1 Bind Request

The function `APSME_BindRequest` initiates the unidirectional bind request. This is a synchronous call.

Prototype

```
zbStatus_t APSME_BindRequest(zbApsmeBindReq_t* pBindReq);
```

Returns

- gZbSuccess_t if binding worked
- gZbIllegalDevice_t if the short or long address is not valid
- gZbIllegalRequest_t if the device is not on a network
- gZbTableFull_t if the table is full
- gZbNotSupported_t if binding is not supported

5.3.2 Unbind Request

This function `APS_UnbindRequest` unbinds, or breaks the logical link between devices. This is an in-memory association only, with no over-the-air behavior.

Prototype

```
zbStatus_t APSME_UnbindRequest(zbApsmeBindReq_t* pBindReq);
```

Returns

- gZbSuccess_t if it worked
- gZbIllegalDevice_t if the short or long address is not valid
- gZbIllegalRequest_t if the device is not on a network
- gZbTableFull_t if the table is full
- gZbNotSupported_t if binding is not supported

5.3.3 Find Binding Entry

The function `APS_FindBindingEntry` looks through the binding table for the entry described by *pBindEntry. The cluster for this helper function is ignored for matching.

Prototype

```
bindIndex_t APS_FindBindingEntry(zbApsmeBindEntry_t* pBindEntry);
```

Returns

- Index into binding table if entry exists
- gApsNotInBindingTable_c if not found

5.3.4 Find Next Binding Entry

The function `APS_FindNextBindingEntry` is used internally by the APSDE-DATA.request primitive. This function requests the next binding entry based on the source endpoint, and start index.

Prototype

```
bindIndex_t APS_FindNextBindingEntry(bindIndex_t iStartIndex, zbEndPoint_t srcEndPoint,
zbIeeeAddr_t aExtAddr);
```

Returns

- Index to the binding entry
- gApsNotInBindingTable_c if not found

5.3.5 Clear Binding Table

The `APS_ClearBindingTable` function call clears every entry in the binding table.

Prototype

```
void APS_ClearBindingTable(void);
```

The function call returns no value.

5.3.6 Add Group Request

Nodes may have multiple endpoints. Before adding an endpoint to a group, the endpoint must be a registered endpoint on its node**.** Note that the endpoint is an endpoint number (1-240), not an index into the endpoint array (0-n).

The function `APSME_AddGroupRequest` adds an endpoint to a specified group.

Prototype

```
zbStatus_t APSME_AddGroupRequest(zbApsmeAddGroupReq_t *pRequest);
```

Returns

- gZbSuccess_c, if it worked
- gZbTableFull_c, if the group table is full

For more information, see APSME-ADD-GROUP.request in the *ZigBee Specifications, r13*.

5.3.7 Remove Group Request

Remove the endpoint from the group using `APSME_RemoveGroupRequest`.

Prototype

```
zbStatus_t APSME_RemoveGroupRequest(zbApsmeRemoveGroupReq_t *pRequest);
```

Returns

- gZbSuccess_c, if removal succeeded
- gZbNoMatch_c, if group invalid or endpoint not part of group

See APSME-REMOVE-GROUP.request in Section 2.2.4.5.3, *ZigBee Specifications, r13*.

5.3.8 Remove Endpoint from All Groups Request

Remove a given endpoint from all groups with APSME_RemoveAllGroupsRequest.

Prototype

```
zbStatus_t APSME_RemoveAllGroupsRequest(zbApsmeRemoveAllGroupsReq_t *pRequest);
```

Returns

- gZbSuccess_c, if removal succeeded
- gZbInvalidEndpoint_c, if removal failed

NOTE

To remove all groups, call ApsGroupReset().

For more information, see APSME-REMOVE-ALL-GROUPS.request in the *ZigBee Specifications, r13*.

5.3.9 Identify Endpoint Group Membership

This internal function confirms that the endpoint is a member of a specified group.

Prototype

```
bool_t ApsGroupIsMemberOfEndpoint(zbGroupId_t aGroupId, zbEndPoint_t endPoint);
```

Returns

It returns TRUE or FALSE.

5.3.10 Group Reset Function

This function resets or removes all groups.

Prototype

```
void ApsGroupReset(void);
```

Returns

This function does not return a value.

5.4 AIB Attributes

The attributes shown in Table 5-1 manage the APS layer in BeeStack.

Table 5-1. APS Information Base Attributes

Attribute	ID	Type	Range	Description	Default
apsAddressMap	0xc0	Set	Variable	Current set of 64 bit IEEE to 16 bit NWK address maps	Null set
apsBindingTable	0xc1	Set	Variable	Current set of binding table entries in the device	Null set
apsGroupTable	0x0c2	Set	Variable	Current set of group table entries	Null set

Chapter 6
ZigBee Device Objects

ZigBee device objects (ZDO) provide an interface between the application objects, the device profile, and the APS layer. As part of the application layer, ZDO meets common requirements of all applications operating in BeeStack.

ZDO responsibilities include:

- Initializing the APS and NWK layers and the Security Service Provider (SSP).
- Assembling configuration information from the end applications to determine and implement discovery, security management, network management, and binding management.

The ZDO interface utilizes the APSDE-SAP for data and the APSME-SAP for control messages.

ZDO presents public interfaces to the application objects in the AF layer for control of device and network functions by the application objects. ZDO communicates with the lower portions of the ZigBee protocol stack on endpoint 0.

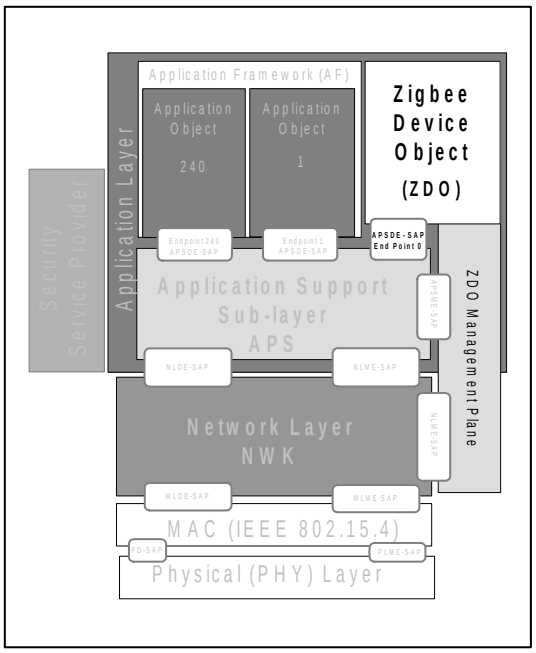

Figure 6-1. ZigBee Device Objects in BeeStack

6.1 ZDO State Machine

The ZDO state machine process is automated. The descriptions provided here clarify the behavior of the devices. Most important are these general macros and functions for starting and stopping the state machine.

If the non volatile memory (NVM) module is enabled, some of the data gathered for the device configuration is stored in NVM. In order to recover the information following a device reset, use the ZDO macro ZDO_StartWithNWM.

For more information, see the comments included in the `ZdoStateMachineHandler.c` file.

6.2 General ZDO Interfaces

This section includes the general ZDO macros and functions to all devices, regardless of their role (ZC, ZR, or ZED).

6.2.1 Get State Machine

This macro retrieves the current state of the ZDO machine. The states for ZDO are defined in `ZdoCommon.h`. Generally, this function is not needed since the change in ZDO state is reported to ASL through the use of ASL_ZdoCallBack(). If not using ASL, the application can register to receive ZDO state change information using Zdp_AppRegisterCallBack().

Macro

#define ZDO_GetState()(gZDOState)

The following states for ZDO are defined in BeeStack (internal use only states are not shown):

gZdoDiscRetryState	Retrying network discovery
gZdoReadyState_c	ZDO ready, and stopped.
gZdoDiscInProgressState_c	Network discovery in progress
gZdoFormationState_c	Network formation in progress
gZdoJoiningInProgressState_c	Joining a network
gZdoLeaveInProgressState_c	Leaving a network
gZdoIdleState_c	Nothing for ZDO to do
gZdoPermitJoin_c	Permit joining has changed state
gZdoOrphanJoinState_c	Ophan joining
gZdoCoordinatorStartingState_c	ZC is starting, but not yet running
gZdoCoordinatorRunningState_c	ZC is running
gZdoRouterRunningState_c	ZR is running
gZdoEndDeviceRunningState_c	ZED is running
gZdoDeviceAuthenticationState_c	Device is being authenticated
gZdoStopState_c	ZDO has stopped

gKeyTransferState_c	Key is being transferred.

6.2.2 Start ZDO State Machine without NVM

The macro `ZDO_Start(gStartWithoutNWM_c)` initializes the device using the default values.

Macro

```
#define ZDO_Start(gStartWithOutNvm_c)
```

6.2.3 Start ZDO State Machine with NVM

Starting the ZDO state machine with NVM recovers all of the values from memory; for example, neighbors, routes, and other stored information.

Macro

```
#define ZDO_Start (startMode)
```

The start mode parameter can be any one of the following:

gStartWithOutNvm_c	Allows a device to join the network fresh, without restoring NVM.
gStartAssociationRejoinWithNvm_c	Allows a device to rejoin the network with the association procedure using the PAN information from its memory.
gStartOrphanRejoinWithNvm_c	Allows a device to rejoin the network with the orphan procedure using the information from its memory.
gStartNwkRejoinWithNvm_c	Allows a device to rejoin the network at the NWK layer with the information from its memory using rejoin command.
gStartSilentRejoinWithNvm_c	Allows a device to rejoin a network, and restores information from its memory, without notifying other devices of its return to the network.

6.2.4 Stop ZDO State Machine

This macro instructs a device to stop its ZDO State machine.

Macro

```
#define ZDO_Stop()
```

6.2.5 Stop ZDO and Leave

This macro sends to a device the command to leave the network and then stop its ZDO state machine.

Macro

```
#define ZDO_Leave()
```

6.3 Device Specific ZDO Interfaces

These ZDO macros and functions, while specific to the ZC, ZR, or ZC, again are automated, and the information that follows describes their behavior.

For each device, there are events that are supported only for the specific state. For example, when a ZC is in running state, it cannot process a start event.

6.3.1 ZC State Machine

The ZC state machine supports several events depending upon its state.

This macro can change its state:

`ZDOCoordinatorChangeState(state)`

When in any of the following states, there are limited events supported for the coordinator.

6.3.1.1 ZC Initial State

When an application starts, restarts or resets a coordinator, it enters initial machine state and restores any required information from NVM.

When in initial machine state, the ZC machine state supports these events:

- gStartAssociationRejoinWithNvm_c
- gStartOrphanRejoinWithNvm_c
- gStartNwkRejoinWithNvm_c
- gStartSilentRejoinWithNvm_c

6.3.1.2 ZC Starting State

The ZC enters starting state following initial state, and after restoring any required information from NVM.

When in starting machine state, the ZC machine state supports:

- gZDO_StartNetworkFormation_c
- gZDO_NetworkFormationSuccess_c
- gZDO_NetworkFormationFailed_c
- gZDO_Timeout_c

6.3.1.3 ZC Running State

In running machine state, the ZC machine state supports:

- gStop_c
- gKeyTransferInitiated_c
- gManagementCommandSent_c
- gChildLeaveSuccess_c

6.3.1.4 ZC Key Transfer state

This machine state supports the following event only with the key transfer initialized.

- gKeyTransferSuccess_c

6.3.1.5 ZC Stop State

A ZC enters stop state when network formation fails or an application tells the device to stop. Additionally, this clears the stack and NVM.

The ZC stop state supports this event:

- gZdoStopState_c

6.3.1.6 ZC Remote Commands State

The ZC enters the remote command state when the device receives any remote command. A ZC moves to running state upon receipt of the remote command.

- gZdoRemoteCommandsState_c

6.3.2 ZR State Machine

The router state machine supports several events depending upon its state.

This macro can change its state:

```
ZDORouterChangeState(state)
```

The router can be in any of the following states. When in any given state, there are limited events that are supported.

6.3.2.1 Initial Machine State

When in initial machine state, the ZR machine supports these events:

- gStartWithOutNvm_c
- gStartAssociationRejoinWithNvm_c
- gStartOrphanRejoinWithNvm_c
- gStartNwkRejoinWithNvm_c
- gStartSilentRejoinWithNvm_c

In the case of StartSilentRejoinWithNvm, a device can join the network and restore information from its memory without notifying other devices of its return to the network

6.3.2.2 Discovery in Progress State

If in discovery-in-progress state, the ZR machine state supports these events:

- gZDO_StartNetworkDiscovery_c
- gZDO_NetworkDiscoverySuccess_c
- gZDO_NetworkDiscoveryFailed_c

6.3.2.3 Joining In Progress State

If in joining-in-progress state, the ZR machine state supports these events:

- gZDO_StartJoiningNetwork_c
- gZDO_StartRouterSuccess_c
- gZDO_JoinFailed_c

6.3.3 ZED Machine State

This macro can change the ZED machine state:

`ZDOEnddeviceChangeState(state)`

When in any of the following states, there are limited events supported for an end device.

6.3.3.1 ZED Initial State

An application starts, restarts, or resets a ZED, which triggers initial machine state and restores all information from NVM, if required.

When in initial machine state, the ZED state machine supports these events:

- gStartAssociationRejoinWithNvm_c
- gStartOrphanRejoinWithNvm_c
- gStartNwkRejoinWithNvm_c
- gStartSilentRejoinWithNvm_c

6.3.3.2 ZED Discovery In Progress State

Following initial machine state, a ZED enters discovery-in-progress state and seeks out a parent device so that it can join the network by the discovery process.

The ZED discovery-in-progress machine state supports the following events:

- gZDO_StartNetworkDiscovery_c
- gZDO_NetworkDiscoverySuccess_c
- gZDO_NetworkDiscoveryFailed_c
- gZDO_TimeoutBetweenScan_c
- gZDO_Timeout_c

6.3.3.3 ZED Joining In Progress State

If the ZED succeeded in discovering a network then it enters a joining-in-progress machine state.

This ZED machine state supports the following events:

- gZDO_StartJoiningNetwork_c
- gZDO_JoinSuccess_c
- gZDO_JoinFailed_c
- gAuthenticationInitiated_c

6.3.3.4 ZED Orphan Join State

A ZED may try to join the network by the orphan scan process after the initial state.

When in this state, the ZED orphan join state supports these events:

- gZDO_StartOrphanJoin_c
- gZDO_JoinSuccess_c
- gZDO_JoinFailed_c
- gZDO_Timeout_c

6.3.3.5 ZED Running State

When a join or authentication (in a secured network) request succeeds, a ZED enters running state.

ZED running state supports these events:

- gStartDevice_c
- gStartNwkRejoinWithNvm_c
- gStop_c
- gAnnceStop_c
- gManagementCommandSent_c

6.3.3.6 ZED Leave-In-Progress State

A ZED enters the leave-in-progress state when it initiates or receives a leave request, or if it receives the stop request from the application.

ZED leave-in-progress state supports the following events:

- gZDO_DeviceLeftNetwork_c
- gZDO_DeviceLeftNetwork_c
- ZDO_StartLeaving_c

6.3.3.7 ZED Stop State Machine

A ZED enters stop state when discovery, join, or authentication fails, or an application sends a stop request. Additionally, this clears the stack and NVM.

ZED stop state machine supports the following event:

- gStop_c

6.3.3.8 ZED Authentication State

A ZED enters device authentication state once the connection to a network succeeds with security enabled. This applies to residential security mode only.

Device authentication state for the ZED supports the following events:

- gZdoDeviceAuthenticationState_c
- gAuthenticationSuccess_c
- gAuthenticationFailure_c
- gZDO_DeviceLeftNetwork_c

6.3.3.9 ZED Remote Command State

A ZED enters the remote command state when the device receives any remote command. The ZED moves to running state upon receipt of the remote command.

The ZED remote command state supports the following event:

- gZDO_MgmtResponseSent_c

6.4 Selecting PAN ID, Channel and Parent

The choice of PAN ID, channel and parent are all under application control for all BeeStack nodes.

By default, BeeStack uses the following algorithm to select a PAN ID and Channel when forming a network (ZC only):

- The set of channels is defined by mDefaultValueOfChannel_c
- Of the set of channels, look for the channel with the fewest network
- Of the channels with the fewest networks find the channel with the least noise
- Choose use the MAC address for extended PAN ID if mDefaultNwkExtendedPANID_c is all 0x00s, otherwise use mDefaultNwkExtendedPANID_c for the extended PAN ID
- Choose a random 16-bit PAN ID if mDefaultValueOfPanId_c is 0xff, 0xff, otherwise use mDefaultValueOfPanId_c
- Do not form the network if the PAN ID (extended or 16-bit) is already in use. Use a random PAN

By default, BeeStack uses the following algorithm to select an appropriate parent and channel when joining a network (ZC and ZED only):

- The set of channels is defined by mDefaultValueOfChannel_c

- In the set of channels, look for at least one node from each network, up to the limit of gathered nodes (depends on size of neighbor table)
- If there is more than one node in a given network, chose the one that is highest up the tree with both capacity and joining enabled. Each router has by default the capacity for 6 router children and 14 end device children
- Choose the first node that meets the above criteria for a parent and request association

These algorithms can be modified by the application, but it takes good knowledge of C programming. The algorithms are in the file AppStackImpl.c, and use the following functions for channel, PAN and parent selection:

```
void SelectLogicalChannel
(
  const nwkMessage_t *pMsg, /* IN – energy detect scan confirm */
  uint8_t* pScanChannels,/* IN - list of channels obtained */
uint8_t* pSelectedLogicalChannel /* IN/OUT- To be updated after
                                            finding least number of Nwks */
);
void SelectPanId
(
  const nwkMessage_t *pMsg, /* IN -active scan confirm */
  uint8_t selectedLogicalChannel, /* IN - Channel */
  uint8_t* pPanId/* IN/OUT - Pointer to the PanId */
);
index_t SearchForSuitableParentToJoin ( void );
```

Chapter 7
ZigBee Device Profile

The ZigBee Device Profile (ZDP), found within ZDO, describes how ZDO implements features such as service discovery and device discovery, end device bind and unbind, and binding table management. Within ZDP, clusters and device descriptions define the supported ZigBee device capabilities.

The ZDO profile resides on endpoint zero, while all other application endpoints are assigned endpoints 1 through 240.

ZDP offers service primitives for device and service discovery, binding, and network management for the client and server activities.

This section includes all information and service requests. Since BeeStack automatically generates the ZDP response, if any, they are not described here. For more information, see the ZigBee Specification, revision 13, for more information.

7.1 Application Support Layer

BeeStack augments the communication capabilities of ZDO with features available from the BeeStack application Support layer (ASL). While not a true layer, the ASL generates commands in a form that ZDP can efficiently process. This BeeStack element serves as a support layer for the common user interface for applications, including ZCL and ZDO, for example.

ASL uses the SAP handlers to send commands to ZDP. For example, the ASL_NWKAddr_req is sent to the ZDP for processing (see NWK_addr_req). For more information about ASL functions, see Chapter 5, “Application Support Sub-layer”.

Every ZDP function may be enabled or disabled through BeeKit properties or by enabling or disabling the option in `BeeStackConfiguration.h`. If BeeKit is used to adjust the property, the entire project does not need to be exported: a simple export properties will suffice. If modifying the property in the source code, set it to TRUE to enable the command, and FALSE to disable the command.

For example, to enable NWK_addr_req in `BeeStackConfiguration.h`, use:

```
#define gNWK_addr_req_d  TRUE
```

Requests and responses are enabled or disabled separately. In the ZigBee specification, responses can be mandatory while the request is optional. By separating the requests and responses, the developer can choose which commands are appropriate for a given application, saving code space by disabling those which are not required.

When an option is disabled, the ASL request function is stubbed via a C macro. This allows the C code to continue to compile without error when enabling and disabling various options, but be aware that the ASL code for that request no longer functions. For example, if an application has disabled gNWK_addr_req_d, but calls on ASL_NWK_addr_req(), as shown in the code below:

```
…
  (void)ASL_NWK_addr_req(NULL, aDestAddress, aIeeeAddr, 0, 0);
…
```

The actual code will be nothing, because the C macro will be defined as an empty macro:

```
#define ASL_NWK_addr_req \ (pSequenceNumber,aDestAddress,aIeeeAddr,requestType,startIndex)
```

NOTE

The ASL in BeeStack should not be confused with ZigBee's application support sub-layer (APS).

7.2 Device and Service Discovery

The distributed operations of device and service discovery allow individual devices or designated discovery cache devices to respond to discovery requests.

The field, device address of interest, enables responses from either the device itself or a discovery cache device. When both the discovery cache device and the device address of interest respond, the response from the device address of interest takes precedence.

7.2.1 Device Discovery

Device discovery enables a device to determine the identity of other devices on the personal area network (PAN). Device discovery supports both 64-bit IEEE addresses and 16-bit network addresses, and it uses broadcast or unicast addressing.

With a broadcast-addressed discovery request, all devices on the network respond according to the logical device type and match criteria.

- ZigBee end devices (ZED) respond with address only
- A ZigBee coordinator (ZC) or ZigBee router (ZR) with associated devices responds with additional information: Their address is the first entry, and it may be followed, depending on the type of request, by the addresses of their associated devices. The responding devices use APS acknowledged service on the unicast response

When unicast addressed, only the specified device responds. A ZED responds with its address only, while a ZC or ZR responds with its own address and the addresses of all associated child devices. The inclusion of the associated child devices allows the requester to determine the underlying network topology for the specified device.

7.2.2 Service Discovery

Service discovery enables a device to determine services offered by other devices on the PAN.

With broadcast address service discovery, only the individual device or primary discovery cache responds with the requested criteria match (due to the volume of information that could be returned if every network node responded). The primary discovery cache responds only if it contains cached discovery information for the NWK address of interest. Also, the responding device responds with unicast APS-acknowledged service.

With unicast addressed service discovery, only the specified device responds. A ZC or ZR must cache the Service Discovery information for sleeping associated devices and respond on their behalf.

This chapter describes the following service discovery commands:

- Active End Point
- Match Simple Descriptor
- Simple Descriptor
- Node Descriptor
- Power Descriptor
- Complex Descriptor
- User Descriptor

7.3 Primary Discovery Cache Device Operation

The node descriptor both configures and advertises a device as a primary discovery cache device. This primary discovery cache device operates as a state machine with respect to clients utilizing its cache services.

The primary discovery cache device has the following states:

- Undiscovered
- Discovered
- Registered
- Unregistered

If undiscovered, the primary discovery cache device uses a radius-limited broadcast, the discovery register request, to all RxOnWhenIdle devices. It attempts to locate a primary discovery cache device within the radius supplied in the request.

When discovered, the client unicasts a request to the discovery cache device, along with the sizes of the discovery cache information it seeks to store. The discovery cache device responds with a SUCCESS or TABLE_FULL.

A registered client is one that has received a SUCCESS status response from the discovery cache device from a previous request. The client then uploads the discovery information using the node, power, active endpoint, and simple descriptor store requests. This enables the primary discovery cache device to fully respond to discovery requests on the client's behalf.

Any client or device can remain unregistered by using the remove node cache request. This removes the device from the primary discovery cache device.

NOTE

When the device holds its own discovery cache, the device then responds to identify itself as the repository of discovery information.

7.4 Binding Services

As described in , binding creates logical links between application device endpoints to allow them to work together to perform specific tasks. Binding maintains information on each logical link in a binding table. Each device in the network keeps its own binding table, although the ZC acts as a broker when end device bind is used.

A primitive initiates a binding operation on a device that supports a binding table. ZDO, the next higher layer, generates this primitive which it issues to the APS sub-layer. This is an in-memory association only with no over-the-air behavior.

Binding is unidirectional between devices. That is, the receiving device sends a response if needed, but issues no binding requests itself. For example, a switch sends a binding request to a light; the light may respond, however, the light sends no binding request on its own.

7.5 ZDP Functions and Macros

ZDP, similar to any ZigBee profile, operates by defining device descriptions and clusters. The device descriptions and clusters in ZDP, however, unlike application-specific profiles, define capabilities supported in all ZigBee devices.

The functions and macros in this section describe some of the key activities required to establish device communication using BeeStack.

7.5.1 ZDP Register Callback

BeeStack uses the register callback messaging function Zdp_AppRegisterCallBack() to register which function will receive ZDO state change information. This does not affect data indications or confirms.

The applications which use ASL (Application Support Library), are set up to receive ZDO state change information in the function ASL_ZdoCallBack() found in file ASL_UserInterface.c. If registered, ZDO informs the application when the node has formed or joined a network, changed its permit join status and the like. See ZDO_GetState() for more information.

Primitive

```
void ZDP_AppRegisterCallBack (ZDPCallBack_t pZdpAppCallBackPtr);
```

Parameters

- pointer to the response function

7.5.2 ZDP NLME Synchronization Request

Use this function to manually poll a parent for data from a ZigBee End Device (ZED). This is used in low power modes, so the device can sleep for long periods, then poll the parent when it wakes up. This is used automatically by applications using ASL.

Primitive

```
void ASL_Nlme_Sync_req(bool_t track);
```

Parameters

The track parameter is ignored and should be set to FALSE.

7.6 Device and Service Discovery – Client Services

The commands that follow are unicast or broadcast addressed depending on their intent. A request for a device address is broadcast while the requester searches. Devices will unicast the response, since only the requester needs the information. All client side services are optional.

Table 7-1. Device and Discovery Commands

Client Service	Cluster ID
ASL_NWK_addr_req	0x0000
ASL_IEEE_addr_req	0x0001
ASL_Node_Desc_req	0x0002
ASL_Power_Desc_req	0x0003
ASL_Simple_Desc_req	0x0004
ASL_Active_EP_req	0x0005
APP_ZDP_MatchDescriptor	0x0006
ASL_Complex_Desc_req	0x0010
ASL_User_Desc_req	0x0011
ASL_Discovery_Cache_req	0x0012
ASL System_Server_Discovery_req	0x0015
ASL_Discovery_store_req	0x0016
ASL_End_Device_annce	0x0013
ASL_User_Desc_set	0x0014
ASL_Discovery_store_req	0x0016
ASL_Node_Descr_store_req	0x0017
ALS_Power_Des_store_req	0x0018
ASL_Active_EP_store_req	0x0019
ASL_Remove_node_cache_req	0x001b
ASL_Find_node_cache_req	0x001c

7.6.1 Network Address Request

A local device generates the ASL_NWK_add_req when seeking the 16-bit address of a remote device based on a known IEEE address. The local device broadcasts the address request to all devices in RxOnWhenIdle state.

This function generates a ZDP NWK_addr_req and passes it to the ZDO layer through the APP_ZDP_SapHandler function.

Prototype

```
void ASL_NWK_addr_req(zbCounter_t  *pSequenceNumber, zbNwkAddr_t  aDestAddress, zbIeeeAddr_t
aIeeeAddr, uint8_t  requestType, index_t  startIndex);
```

ZDP Returns

- Device discards the request and does not generate a response if no match found.
- Remote device generates a response from the request type if match between the contained IEEE address and its own IEEE address (or one held in the discovery cache) found.

7.6.2 IEEE Address Request Command

A device that generates the ASL_IEEE_addr_req requests the IEEE address and compares that address to its local IEEE address or any IEEE address in its local discovery cache.

Prototype

```
void ASL_IEEE_addr_req(zbCounter_t  *pSequenceNumber, zbNwkAddr_t  aDestAddress, zbNwkAddr_t
aNwkAddrOfInterest, uint8_t  requestType, index_t  startIndex);
```

ZDP Returns

- Device discards the request and does not generate a response if no match found.
- Remote device generates a response from the request type if match between the contained IEEE address and its own IEEE address (or one held in the discovery cache) found.

7.6.3 Node Descriptor Request

The command ASL_Node_Desc_req permits an enquiring device to request the node descriptor from the specified device.

Addressed

Unicast

Prototype

```
void ASL_Node_Desc_req(zbCounter_t *pSequenceNumber, zbNwkAddr_t aDestAddress);
```

Returns

- Node descriptor

7.6.4 Power Descriptor Request

The command, ASL_Power_Desc_req, permits an enquiring device to return the power descriptor from the specified device.

Addressed

Unicast

Prototype

```
void ASL_Power_Desc_req(zbCounter_t *pSequenceNumber, zbNwkAddr_t aDestAddress);
```

ZDP Returns

Power descriptor

7.6.5 Simple Descriptor Request

This ASL_Simple_Desc_req command returns the simple descriptor for a supplied endpoint to an enquiring device.

Addressed

Unicast

Prototype

```
void ASL_Simple_Desc_req(zbCounter_t *pSequenceNumber, zbNwkAddr_t aDestAddress, zbEndPoint_t
endPoint);
```

ZDP Returns

Simple descriptor

7.6.6 Active Endpoint Request

The ASL_Active_EP_req command requests information about active endpoints. An active endpoint is an endpoint with an application supporting a single profile, described by a simple descriptor.

Addressed

Broadcast or Unicast

Prototype

```
void ASL_Active_EP_req(zbCounter_t *pSequenceNumber, zbNwkAddr_t aDestAddress);
```

ZDP Returns

Simple descriptor for active endpoint

7.6.7 Match Descriptor Request

The match simple descriptor command, APP_ZDP_MatchDescriptor, allows devices to supply information and ask for information in return.

Addressed

Broadcast or unicast to all RxOnWhenIdle devices

Prototype

```
void APP_ZDP_MatchDescriptor(zbCounter_t *pSequenceNumber, zbNwkAddr_t aDestAddress,
zbSimpleDescriptor_t *pSimpleDescriptor);
```

ZDP Returns

- Profile ID
- Optionally lists of input and/or output cluster IDs
- Identity of an endpoint on the destination device matching the supplied criteria
- In the case of broadcast requests, the responding device uses APS-acknowledged service on the unicast response

7.6.8 Complex Descriptor Request

The complex descriptor is an optional command, unicast-addressed from the device seeking the complex descriptor from a specified device, using the command ASL_Complex_Desc_req.

Prototype

```
void ASL_Complex_Desc_req(zbCounter_t *pSequenceNumber, zbNwkAddr_t aDestAddress);
```

7.6.9 User Descriptor Request

A remote device receives the `ASL_User_Desc_req`, responding with a unicast simple descriptor to the originator of the command.

Addressed

Unicast to originator

Prototype

```
void ASL_User_Desc_req(zbCounter_t *pSequenceNumber, zbNwkAddr_t aDestAddress);
```

ZDP Returns

- SUCCESS status notification with the requested user descriptor in the UserDescriptor field
- Error otherwise, with no UserDescriptor field

7.6.10 Discovery Cache Request

The command ASL_Discovery_Cache_req asks for Remote Devices which are Primary Discovery Cache devices (as designated in their Node Descriptors). Devices not designated as primary discovery cache devices should not respond to the cache discovery command.

Prototype

```
void ASL_Discovery_Cache_req(zbCounter_t *pSequenceNumber, zbNwkAddr_t aDestAddress,
zbNwkAddr_t aNwkAddrOfInterest, zbIeeeAddr_t aIEEEAddrOfInterest);
```

ZDP Returns

- SUCCESS, and the local device uses the Discovery_Store_req (targeted to the remote device supplying the response) to determine if there is sufficient discovery cache storage available.
- The Discovery Cache Request is broadcast at the default broadcast radius (2 * nwkMaxDepth, which defaults to 10 in stack profile 0x01).

7.6.11 End Device Announce

End-Device-Announce request is used to indicate that a node has moved to a new NWK short address (happens automatically by BeeStack) or now has different MAC capabilities (for example, an end-device which has been plugged in can now indicate it is mains powered with RxOnIdle=TRUE). This request should be broadcast to the entire network so any node which communicates with this node can update its internal information.

Prototype

```
void ASL_End_Device_annce(zbCounter_t *pSequenceNumber, zbNwkAddr_t aDestAddress, zbNwkAddr_t
aNwkAddress, zbIeeeAddr_t  aIeeeAddress, macCapabilityInfo_t capability);
```

When the ZC receives the `End_Device_annce` message, it checks the supplied address for a match using binding tables holding 64-bit IEEE addresses for devices within the PAN.

After checking the Binding Table and Trust Center tables and finding a match, the ZC updates its AIB address map entries to reflect the updated 16 bit NWK address contained in the `End_Device_annce`.

7.6.12 User Descriptor Set Request

This optional user descriptor command, ASL_User_Desc_set, permits an enquiring device to get the User Descriptor from the specified device. It is always unicast addressed.

Prototype

```
void ASL_User_Desc_set(zbCounter_t *pSequenceNumber, zbNwkAddr_t aDestAddress, zbSize_t
length, zbUserDescriptor_t aUserDescription);
```

7.6.13 Server Discovery Request

ASL_System_Server_Discovery_req is generated from a Local Device seeking the location of a particular system server or servers as indicated by the ServerMask parameter. The destination addressing on this request is broadcast to all RxOnWhenIdle devices.

Prototype

```
void ASL_System_Server_Discovery_req(zbCounter_t  *pSequenceNumber, zbNwkAddr_t aDestAddress,
zbServerMask_t aServerMask);
```

When a remote device receives this request, it compares the ServerMask parameter to the server mask field in its own Node descriptor.

ZDP Returns

- If any matching bits are found, the remote device sends a System_Server_Discovery_rsp back to the originator using unicast transmission (with acknowledgement request) indicating the matching bits.
- If no matching bits are found, no action is taken.

7.6.14 Discovery Cache Storage Request

The `Discovery_store_req` allows a device on the network to request storage of its discovery cache information on a Primary Discovery Cache device. This request includes the amount of storage space the local device requires, and stores information for replacing a device or a sleeping device.

Prototype

```
void ASL_Discovery_store_req(zbCounter_t *pSequenceNumber, zbNwkAddr_t aDestAddress,
zbDiscoveryStoreRequest_t *pDiscoveryStore);
```

Returns

- gZdoNotSupported_c, if the remote device is not a Primary Discovery Cache device.
- Determines if it has storage for the requested discovery cache size, if the remote device is a primary discovery cache device, by summing the sizes of the these fields:
  - NWKAddr
  - IEEEAddr

  - NodeDescSize
  - PowerDescSize
  - ActiveEPSize
  - sizes from the SimpleDescSizeList
- gZbSuccess_c, if sufficient space exists, and the remote device reserves the storage space requested.
- gZdoTableFull_c, if there is no available space.

Additionally, the Remote Device replaces the previous entry and discovery cache information with the newly registered data if the local device IEEEAddr matches, but the NWKAddr differs from, a previously stored entry.

7.6.15 Store Node Descriptor on Primary Cache Device

A device requests the storage of its node description on a primary discovery cache device using the ASL_`Node_Descr_store_req`. The request includes the information, in this case, the node descriptor, that the local device is attempting to place in cache.

Prototype

```
void ASL_Node_Desc_store_req(zbCounter_t *pSequenceNumber, zbNwkAddr_t aDestAddress,
zbNodeDescriptor_t *pNodeDescriptor);
```

Returns

- gZdoNotSupported_c, if the remote device is not a primary discovery cache device.
- gZbSuccess_c, if the NWKAddr and IEEEAddr in the request referred to addresses already held in the Primary Discovery Cache, the descriptor in this request shall overwrite the previously held entry.
- gZdoInvalidRequestType_c, if ASL_Discovery_store_req() was not successfully called for this node.

7.6.16 Store Power Descriptor Request

Similarly, the function call `ASL_Power_Desc_store_req` seeks to store the local device's power information in a remote device's primary discovery cache.

Prototype

```
void ASL_Power_Desc_store_req(zbCounter_t *pSequenceNumber, zbNwkAddr_t aDestAddress,
zbPowerDescriptor_t *pPowerDescriptor);
```

Returns

- gZdoNotSupported_c, if the remote device is not a primary discovery cache device
- gZbSuccess_c, if worked

- gZdoInvalidRequestType_c, if ASL_Discovery_store_req() was not successfully called for this node

7.6.17 Active Endpoint List Storage Request

ASL_Active_EP_store_req enables devices in the network to request storage of their list of active endpoints to a primary discovery cache device that has previously received a SUCCESS status from a Discovery_store_req to the same Primary Discovery Cache device.

Included in this request is the count of Active Endpoints the Local Device wishes to cache and the endpoint list itself.

Prototype

```
void ASL_Active_EP_store_req(zbCounter_t *pSequenceNumber, zbNwkAddr_t aDestAddress,
zbCounter_t activeEPcount, zbEndPoint_t *pActiveEPList);
```

Returns

- gZdoNotSupported_c, if it is not a Primary Discovery Cache device.
- gZbSuccess_c, if storage completed.
- gZdoInvalidRequestType_c, if ASL_Discovery_store_req() was not successfully called for this node.
- If the request returned a status Success and both the NWKAddr and IEEEAddr are already in the primary discovery cache, the remote device replaces the previous entry and discovery cache information with the newly registered data.

7.6.18 Simple Descriptor Storage Request

A device requests the storage of its simple descriptor on a primary discovery cache device using the ASL_Simple_Descr_store_req. This conditional request must come from a node that has previously received a SUCCESS status from an earlier discovery storage request to the same primary discovery cache device.

Prototype

```
void ASL_Simple_Desc_store_req(zbCounter_t *pSequenceNumber, zbNwkAddr_t  aDestAddress,
zbNodeDescriptor_t *pNodeDescriptor);
```

Returns

- NOT_SUPPORTED, if the remote device is not a primary discovery cache device.
- SUCCESS, if the IEEEAddr in the request referred to addresses already held in the primary discovery cache; the descriptor in this request overwrites a previously held entry.
- NOT_PERMITTED, if it has not previously allowed the request.
- INSUFFICIENT_SPACE, if no space to store the simple descriptor.

7.6.19 Remove Node Cache Request

With ASL_Remove_node_cache_req, ZigBee devices on the network request that a Primary Discovery Cache device remove the discovery cache information for a specified ZED.

This request undoes a previously successful Discovery_store_req and additionally removes any cache information stored on behalf of the specified ZED on the Primary Discovery Cache device.

Prototype

```
void ASL_Remove_node_cache_req(zbCounter_t *pSequenceNumber, zbNwkAddr_t aDestAddress,
zbNwkAddr_t aNwkAddress, zbIeeeAddr_t aIeeeAddress);
```

ZDP Returns

- NOT_SUPPORTED, if not a primary discovery cache device.
- NOT_PERMITTED, if a prior response with anything but SUCCESS was issued.
- SUCCESS, if primary discovery cache device, and overwrites all cached discovery information for the device of interest.

7.6.20 Find Node Cache Request

The ASL_Find_node_cache_req() allows a ZigBee node to find which node in the network is caching information for the requested node. Both the IEEE and NWK address must match the entry to be found. The aDestAddress should be set to gaBroadcastRxOnIdle to find the proper cache.

Prototype

```
void ASL_Find_node_cache_req(zbCounter_t *pSequenceNumber, zbNwkAddr_t aDestAddress,
zbNwkAddr_t aNwkAddress, zbIeeeAddr_t aIeeeAddress);
```

ZDP Returns

- gZbSuccess_c if found, and the return address in the response function as registered by Zdp_AppRegisterCallBack().
- No response will be received if there is no cache in the network that supports the node in question.

7.7 Binding Management Service Commands

The requests to bind to a device, as well as store, back up, or recover binding table entries, are unicast to a destination device. The list in Table 7-2 includes the unicast-addressed commands and cluster IDs for the commands detailed in the sections that follow.

Many commands in this section use an "index" as one of the parameters. Sending or retrieving tables, the size of the table may exceed the maximum size of a ZigBee packet. In this case, a partial list is sent over the air. The index is used to indicate where in the list this partial list begins. For example, if ASL_Backup_Bind_Table_req(), is issued, the first time it would be called with a StartIndex of 0 . If only 6 binding table entries can fit in the payload, then the next time it is called, the StartIndex would be set to 6, and so on through the table. The number of entries in the partial list that may be sent over the air depends

on the size of the structure in question. When receiving entries from a table, BeeStack will automatically calculate the proper size. When the application is transmitting a table, the maximum size can be calculated by using the maximum payload of 80 bytes, subtracting the header for that payload, and dividing by the size of each entry.

Table 7-2. Service Commands for Binding Management

Command	Cluster ID
APP_ZDP_EndDeviceBindRequest	0x0020
APP_ZDP_BindRequest	0x0021
APP_ZDP_UnbindRequest	0x0022
ASL_Bind_Register_req	0x0023
ASL_Replace_Device_req	0x0024
ASL_Store_Bkup_Bind_Entry_req	0x0025
ASL_Remove_Bkup_Bind_Entry_req	0x0026
ASL_Backup_Bind_Table_req	0x0027
ASL_Recover_Bind_Table_req	0x0028
ASL_Backup_Source_Bind_req	0x0029
ASL_Recover_Source_Bind_req	0x002a

7.7.1 End Device Bind Request

This command binds two nodes together using a button press or some similar user interface mechanism. This command is always issued to the ZigBee coordinator (ZC), so the aDestAddress must always be { 0x00, 0x00 }. The ZC then determines if the two nodes match (for example, a light and a switch).

If the two nodes match, then they are bound together. A match is determined by comparing the input cluster of one node with the output cluster of the other node. Both nodes are checked for a match. If an input cluster on one side (for example, the OnOffCluster 0x0006) matches the output cluster on the other side (for example, 0x0006), then they are considered a match. The side with the output cluster receives the following binding commands.

NOTE

Both sides may match on the output cluster, in which case both sides would receive the binding commands.

Bindings are actually stored in the nodes themselves (source binding), not in the ZC.

First an UnBindRequest() is issued by the ZC to the matching node, then, if that is successful, a BindRequest(). The reason for this is that EndDeviceBind is a toggle. That is, if the nodes are already bound, then it will unbind. If the nodes are not bound, then it will bind.

Prototype

```
void APP_ZDP_EndDeviceBindRequest(zbCounter_t *pSequenceNumber, zbNwkAddr_t aDestAddress,
zbSimpleDescriptor_t *pSimpleDescriptor);
```

ZDP Returns

- gZdoNoMatch_c, if the two nodes do not match.
- gZdoInvalidEndPoint_c, if the source endpoint is out of range (valid range is 1-240).
- gEndDevBindTimeOut_c, if a second node doesn't request EndDeviceBind
- gZbSuccess_c worked. Devices are either bound or unbound (depending on toggle).

7.7.2 Bind Request

A local device seeking to add a binding table entry generates the ZDP_BindRequest, using the contained source and destination addresses as parameters. The unicast destination address must be that of the Primary binding table cache or the SrcAddress.

Prototype

```
void APP_ZDP_BindRequest(zbCounter_t *pSequenceNumber, zbNwkAddr_t aDestAddress, zbMsgId_t
BindUnbind, zbBindRequest_t *pBindUnBindRequest);
```

ZDP Returns

- NOT_SUPPORTED, if the SrcAddress is specified but binding manager unsupported on the remote device.
- SUCCESS, and SrcAddress added.

7.7.3 Unbind Request

A local device seeking to remove a binding table entry generates the ZDP_UnbindRequest, using the contained source and destination addresses as parameters. The unicast destination address must be that of the primary binding table cache or the SrcAddress.

Prototype

```
void APP_ZDP_UnbindRequest(zbCounter_t  *pSequenceNumber, zbNwkAddr_t aDestAddress, zbMsgId_t
Unbind, zbUnbindRequest_t *pUnBindRequest);
```

ZDP Returns

- NOT_SUPPORTED, if the SrcAddress is specified but the binding manager is unsupported on that remote device.
- NO_ENTRY, if a binding table entry does not exist for the SrvAddress, SrcEndp, ClusterID, DstAddress, and DstEndp contained as parameters.
- SUCCESS, otherwise, and the remote device, which is either a primary binding table cache or the SrcAddess, removes the binding table entry based on the Unbind_req parameters.

7.7.4 Local Bind Register Request

A local device generates the ASL_Bind_Register_req to notify a primary binding table cache device that the local device will hold its own binding table entries. The local device uses the unicast destination address to the primary binding cache device.

Prototype

```
void ASL_Bind_Register_req(zbCounter_t *pSequenceNumber, zbNwkAddr_t aDestAddress,
zbIeeeAddr_t aNodeAddress);
```

ZDP Returns

- NOT_SUPPORTED, if the remote device is not a primary binding table cache.
- SUCCESS, and adds the NodeAddress given by the parameter to its table of other source devices that have chosen to store their own binding table.
- TABLE_FULL if the request fails.

NOTE

If an entry for the NodeAddress already exists in the table of source devices, the behavior will be the same as if it had been newly added. To avoid synchronization problems, the source device should clear its source binding table before issuing this ASL_Bind_Register_req command.

When a `SUCCESS` status message results, any existing bind entries from the binding table with source address NodeAddress are sent to the requesting device for inclusion in its source bind table. See Bind_Register_rsp for additional information on this response.

7.7.5 Replace Device Request

ASL_Replace_Device_req requests that a primary binding table cache device change, as specified, all binding table entries that match OldAddress and OldEndpoint.

NOTE

OldEndpoint = 0 has special meaning and signifies that only the address needs to be matched. In this case, the endpoint in the binding table is not changed and NewEndpoint is ignored.

Processing the ASL_Replace_Device command changes all binding table entries for which the source address is the same as OldAddress. If OldEndpoint is non-zero, this additionally changes to NewEndpoint the binding table entry to for which the source endpoint is the same as OldEndpoint.

It changes all binding table entries for which the destination address is the same as OldAddress (and if OldEndpoint is non-zero) and the destination endpoint the same as OldEndpoint. The destination addressing mode for this request is unicast.

Prototype

```
void ASL_Replace_Device_req(zbCounter_t *pSequenceNumber, zbNwkAddr_t aDestAddress,
zbNwkAddr_t aOldAddress, zbEndPoint_t oldEndPoint, zbNwkAddr_t aNewAddress, zbEndPoint_t
newEndPoint);
```

ZDP Returns

- NOT_SUPPORTED, if the remote device is not a primary binding table cache.
- The primary binding table cache confirms that its OldAddress is non-zero. It then searches its binding table for entries of source addresses and entries, or destination addresses and source addresses, set the same as OldAddress and OldEndpoint.
- In the case that OldEndpoint is zero, the primary binding table cache searches its binding table for entries whose source address or destination address match OldAddress. It changes the address to NewAddress, leaving the endpoint value unchanged and ignoring NewEndpoint.
- SUCCESS, then it changes these entries to have NewAddress and NewEndpoint.

For more information on this command, refer to ZigBee Specification revision 13, December 2006.

7.7.6 Store Backup Bind Entry Request

A local primary binding table cache generates the Store_Bkup_Bind_Entry_req and, by sending to a remote backup binding table cache device, requests backup storage of the entry. It generates this request whenever a new binding table entry has been created by the primary binding table cache. The destination addressing mode for this request is unicast, and this affects one entry only.

Prototype

```
void ASL_Store_Bkup_Bind_Entry_req(zbCounter_t *pSequenceNumber, zbNwkAddr_t aDestAddress,
zbStoreBkupBindEntryRequest_t *pStoreBkupEntry);
```

ZDP Returns

- NOT_SUPPORTED, if the remote device is not a backup binding table.
- SUCCESS, if the contents of the Store_Bkup_Bind_Entry parameters match an existing entry in the binding table cache.
- SUCCESS when the backup binding table simply adds the binding entry to its binding table.
- TABLE_FULL, if there is no room to store the information.

If it is the backup binding table cache, it maintains the identity of the primary binding table cache from previous discovery.

7.7.7 Remove Entry from Backup Storage

A local primary binding table cache generates the `ASL_Remove_Bkup_Bind_Entry_req` request and issues the request to a remote backup binding table cache device to remove the entry from backup storage. ZDP generates this request whenever a binding table entry has been unbound by the primary binding table cache. The destination addressing mode for this request is unicast, and it affects only one entry.

Prototype

```
void ASL_Remove_Bkup_Bind_Entry_req(zbCounter_t *pSequenceNumber, zbNwkAddr_t aDestAddress,
zbRemoveBackupBindEntryRequest_t *pRemoveBkupEntry);
```

ZDP Returns

- NOT_SUPPORTED, if the remote device is not a backup binding table cache.
- INV_REQUESTTYPE, if it does not recognize the sending device as the primary binding table cache.
- SUCCESS, keeping the identity of the primary binding table cache from previous discovery.
- NO_ENTRY, if no entry is found.

7.7.8 Backup Binding Table Request

A local primary binding table cache issues the `Backup_Bind_Table_req` request to the remote backup binding table cache device, seeking backup storage of its entire binding table. The destination addressing mode for this request is unicast.

Prototype

```
void ASL_Backup_Bind_Table_req(zbCounter_t  *pSequenceNumber, zbNwkAddr_t aDestAddress,
zbBackupBindTableRequest_t *pBackupBindTable);
```

ZDP Returns

- NOT_SUPPORTED, if the remote device is not a backup binding table cache.
- INV_REQUESTTYPE, if it does not recognize the sending device as a primary binding table cache.
- TABLE_FULL, if this exceeds its table size; it then fills in as many entries as possible.
- SUCCESS, if all other conditions are met, and the table is effectively truncated at the end of the last entry written by the request.
- Since it is a backup binding table cache, it maintains the identity of the primary binding table cache from previous discovery. Otherwise, the backup binding table cache overwrites its binding table entries, starting with StartIndex and continuing for BindingTableListCount entries.
- Unless it returns TABLE_FULL, the response returns the new size of the table (equal to StartIndex + BindingTableListCount).

7.7.9 Recover Binding Table Request

The Recover_Bind_Table_req is generated from a local primary binding table cache and sent to a remote backup binding table cache device when it wants a complete restore of the binding table. The destination addressing mode for this request is unicast.

Prototype

```
void ASL_Recover_Bind_Table_req(zbCounter_t *pSequenceNumber, zbNwkAddr_t aDestAddress,
index_t index);
```

ZDP Returns

- NOT_SUPPORTED if the remote device is not the backup binding table cache.
- INV_REQUESTTYPE, if it does not recognize the sending device as a primary binding table cache.
- SUCCESS, and the backup binding table cache creates a list of binding table entries from its backup beginning with StartIndex and fits as many entries as possible into a Recover_Bind_Table_rsp command.

7.7.10 Source Binding Table Backup Request

The local primary binding table cache generates a Backup_Source_Bind_req to request backup storage of its entire source table of a remote backup binding table cache device. The destination addressing mode for this request is unicast, and it includes the IEEE address.

Prototype

```
void ASL_Backup_Source_Bind_req(zbCounter_t *pSequenceNumber, zbNwkAddr_t aDestAddress,
zbBackupSourceBindRequest_t *pBkupSourceBindTable);
```

ZDP Returns

- NOT_SUPPORTED, if the remote device is not the backup binding table cache.
- INV_REQUESTTYPE, if it does not recognize the sending device as a primary binding table cache.
- TABLE_FULL, if this exceeds its table size.
- SUCCESS if able to complete the request, and the command truncates the backup table to a number of entries equal to its maximum size or SourceTableEntries, whichever is smaller.
- The backup binding table cache otherwise overwrites the source entries in its backup source table starting with StartIndex and continuing through SourceTableListCount entries.

7.7.11 Recover Source Binding Table Request

A local primary binding table cache generates the Recover_Source_Bind_req to send to the remote backup binding table cache device when it wants a complete restore of the source bind table. The destination addressing mode for this request is unicast.

Prototype

```
void ASL_Recover_Source_Bind_req(zbCounter_t *pSequenceNumber, zbNwkAddr_t aDestAddress,
index_t index);
```

ZDP Returns

- NOT_SUPPORTED, if the remote device is not the backup binding table cache.
- INV_REQUESTTYPE, if it does not recognize the sending device as a primary binding table cache.

- SUCCESS, after it creates a list of source bind table entries from its backup beginning with StartIndex and fits as many entries as possible into a Recover_Source_Bind_rsp command.

7.8 Network Management Services

The network discovery requests occur when an end device, node, or application seeks to join or form a network. These services use both client and server components, since the client (end device) makes the request of a device, and the application object on the server sends a response.

7.8.1 Management Network Discovery Request

A local device requests that a remote device scan and then report back any networks in the vicinity of the initiating device using the command `ASL_Mgmt_NWK_Disc_req`. The unicast addressed request includes several parameters, including channels, duration, and network address.

Prototype

```
void ASL_Mgmt_NWK_Disc_req(zbCounter_t *pSequenceNumber, zbNwkAddr_t aDestAddress,
zbChannels_t aScanChannel, zbCounter_t scanDuration, index_t startIndex);
```

ZDP Returns

Nothing

7.8.2 Management LQI Request

A local device looking to obtain a neighbor list for a remote device issues the `ASL_Mgmt_Lqi_req`, along with the link quality indicator (LQI) values for each neighbor. This command uses unicast addressing, and the destination address can only be a ZC or ZR. ZDP responds with the `ASL_Mgmt_Lqi_rsp` command.

Prototype

```
void ASL_Mgmt_Lqi_req(zbCounter_t *pSequenceNumber, zbNwkAddr_t aDestAddress, index_t index);
```

ZDP Returns

Nothing

The remote device (ZR or ZC) retrieves the entries of the neighbor table and associated LQI values using the NLME-GET.request primitive (for the nwkNeighborTable attribute) and with the Mgmt_Lqi_rsp command reports the resulting neighbor table (obtained via the NLME-GET.confirm primitive).

7.8.3 Routing Discovery Management Request

A local device attempts to retrieve the contents of the routing table from a remote device with this ASL_Mgmt_Rtg_req command. The unicast destination address must be that of the ZR or ZC.

Prototype

```
void ASL_Mgmt_Rtg_req(zbCounter_t *pSequenceNumber, zbNwkAddr_t aDestAddress, index_t index);
```

ZDP Returns

Nothing

The routing table is then acquired via the Mgmt_Rtg_rsp command using the NLME-GET.confirm primitive.

7.8.4 Management Bind Request

A Local Device seeking the contents of the binding table from the remote device generates a Mgmt_Bind_req command. The unicast destination address is a primary binding table cache or source device holding its own binding table. Upon receipt, a remote device (ZC or ZR) obtains the binding table entries from the APS sub-layer via the APSMEGET.request primitive (for the apsBindingTable attribute).

Prototype

```
void ASL_Mgmt_Bind_req(zbCounter_t *pSequenceNumber, zbNwkAddr_t aDestAddress, index_t index);
```

ZDP Returns

Nothing

7.8.5 Management Leave Request

A local device requests that a remote device leave the network using the Mgmt_Leave_req command. Generated by a management application, the Mgmt_Leave_req sends the request to a remote device. The remote device executes the request using the NLME-LEAVE.request using the parameters supplied in the Mgmt_Leave_req. The local device is notified of the results of its attempt to cause a remote device to leave the network.

Prototype

```
void ASL_Mgmt_Leave_req(zbCounter_t *pSequenceNumber, zbNwkAddr_t aDestAddress, zbIeeeAddr_t
aDeviceAddres, zbMgmtOptions_t mgmtOptions);
```

ZDP Returns

Nothing

7.8.6 Management Permit Joining

A local device uses the command Mgmt_Permit_Joining_req to request that a remote device (or devices) permit or disallow association. This sets a flag for every device to true or false.

Generated by a management application or commissioning tool on the local device, the NLME-PERMIT-JOINING.request executes using the PermitDuration parameter supplied by Mgmt_Permit_Joining_req. This request affects the trust center authentication if the remote device is the Trust Center and TC_Significance is set to 1. Addressing may be unicast or broadcast to all RxOnWhenIdle devices.

Upon receipt, the remote device(s) shall issue the NLME-PERMITJOINING. request primitive using the PermitDuration parameter supplied with the Mgmt_Permit_Joining_req command.

Prototype

```
void ASL_Mgmt_Permit_Joining_req(zbCounter_t *pSequenceNumber, zbNwkAddr_t aDestAddress,
zbCounter_t permitDuration, uint8_t TC_Significance);
```

ZDP Returns

Nothing

7.8.7 Management Cache

ZigBee devices in a network obtain the list of ZEDs registered with a primary discovery cache device using the Mgmt_Cache_req command. This is a unicast address to the destination primary discovery cache device, which first determines if it is a primary discovery cache and if it supports this optional request primitive.

Prototype

```
void ASL_Mgmt_Cache_Req(zbCounter_t *pSequenceNumber, zbNwkAddr_t aDestAddress, index_t
index);
```

ZDP Returns

Nothing

7.9 ZDO Layer Status Values

Table 7-3 provides status responses for the commands listed above in this section.

Table 7-3. ZDO Status Values

Macro	ID	Description
gZbSuccess_c	0x00	Indicates request succeeded
gZdoInvalidRequestType_c	0x80	Supplied request type was invalid
gZdoDeviceNotFound_c	0x81	Requested device did not exist on a device following a child descriptor request to a parent

Table 7-3. ZDO Status Values (continued)

gZdoInvalidEndPoint_c	0x82	Supplied endpoint was equal to 0x00 or between 0xf1 and 0xff
gZdoNotActive_c	0x83	Requested endpoint is not described by a simple descriptor
gZdoNotSupported_c	0x84	Requested optional feature is not supported on the target device
gZdoTimeOut_c	0x85	Requested operation timed out
gZdoNoMatch_c	0x86	End device bind request was unsuccessful due to failure to match any suitable clusters
gZdoNoEntry_c	0x88	Unbind request was unsuccessful due to ZC or source device not having an entry in its binding table to unbind
gZdoInsufficientSpace_c	0x8a	Device does not have storage space to support the requested operation
gZdoNotPermited_c	0x8b	Device is not in the proper state to support the requested operation
gZdoTableFull	0x8c	Device does not have table space to support the operation
gZdoSetAddrMapFailure_c	0x8d	Unable to add to the address map

Chapter 8
Network Layer

The BeeStack network (NWK) layer handles the following duties:

- Joining and leaving a network
- Applying security to frames
- Routing frames to their intended destinations
- Discovering and maintaining routes between devices
- Discovering one-hop neighbors
- Storing pertinent neighbor information

For the ZC, the NWK layer specifically handles starting a new network when appropriate, as well as assigning addresses to newly associated devices.

The NWK layer provides both correct operation of the IEEE 802.15.4-2003 MAC sub-layer and a suitable service interface to the application layer. Two service entities interface with the application layer to provide those necessary functionalities, the data service and the management service.

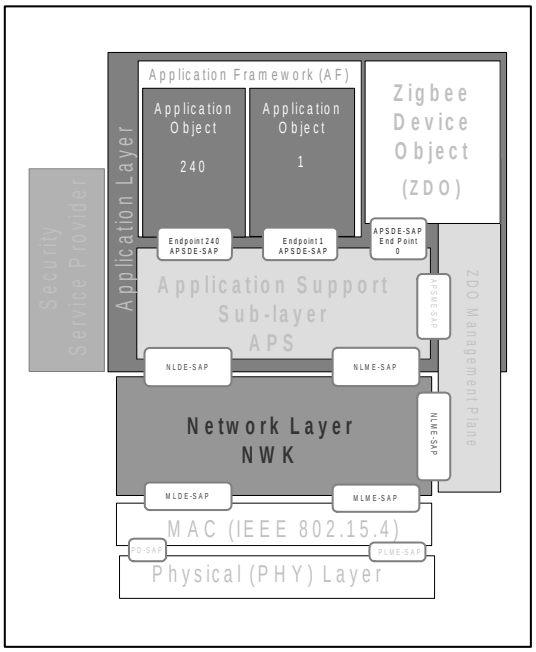

Figure 8-1. Network Layer Interfaces

The NWK layer data entity (NLDE) handles data transmission service through its associated service access point, the NLDE-SAP.

The (NLME) provides data management services through the NLME-SAP. The NLME utilizes the NLDE for some of its management tasks. The NLME also maintains a database of managed objects known as the network information base (NIB).

8.1 Channel and PAN Configuration

These sections describe the channel list and detail how the PAN is configured.

8.1.1 Channel Configuration

The default channel list defines which channels to scan when forming or joining a network.

As shown in Figure 8-2, the channel list is a bitmap, where each bit identifies a channel (for example bit 12 corresponds to channel 12). Any combination of channels can be included. Only channels 11-26 are available to users.

31 28 24 20 16 12 08 04 00		Channel
0000 0000 0000 0000 0000 1000 0000 0000	0x04000000	26
0000 0100 0000 0000 0000 0000 0000 0000	0x00000800	11
0000 0010 0000 0000 0000 0000 0000 0000	0x02000000	25
0000 0111 1111 1111 1111 1000 0000 0000	0x07fff800	All 11-26
0000 0000 1000 0000 0001 0000 0000 0000	0x00800000	23 and 12

Figure 8-2. Channel List Bitmap

8.1.1.1 Channel Default Value

Channel 25 serves as the default network channel value for all applications, although users may change that information using BeeKit.

Macro

```
#define mDefaultValueOfChannel_c
```

Parameter

```
0x02000000
```

Table 8-1. Hexadecimal Channel Values

Channel Number	Channel Value (Hex)	32-bit Value
11	0x0B	0x00000800
12	0x0C	0x00001000
13	0x0D	0x00002000
14	0x0E	0x00004000
15	0x0F	0x00008000
16	0x10	0x00010000

Table 8-1. Hexadecimal Channel Values (continued)

17	0x11	0x00020000
18	0x12	0x00040000
19	0x13	0x00080000
20	0x14	0x00100000
21	0x15	0x00200000
22	0x16	0x00400000
23	0x17	0x00800000
24	0x18	0x01000000
25	0x19	0x02000000
26	0x1A	0x04000000

8.1.2 PAN ID

The personal area network (PAN) ID establishes a unique identifier used to form or join the network. ZigBee PAN IDs range from 0 - 0x3fff (0x00,0x00-0xff,0x3f in the little-endian form that all values are sent over the air).

When forming a network, the PAN ID 0xffff indicates random PAN ID selection. The ZC will generate a PAN ID that does not match any PAN IDs it can locate on its chosen channel. When joining a network, a node with an 0xFFFF PAN ID will join any network it finds that meet any other criteria the application or initial configuration might require.

Macro

`mDefaultValueOfPanId_c`

Default

`0xab, 0x1b`

8.1.3 Beacon Notify

A device issues a beacon request every time it performs network discovery (during network forming and joining). The beacon request goes out over the air to any device in range. Every ZC and ZR that hears the beacon request must return a beacon response.

The MAC layer of the device receiving the beacon response passes the response to the NWK layer as a beacon-notify indication.

8.1.3.1 Parse Beacon Notification

The function ParseBeaconNotifyIndication processes every beacon-notify indication received, with the function exposed to the application so it can filter the beacons. This filtering allows the application to choose the appropriate response to be included in the list to select a router. In the case of a ZC, the filtering checks for PAN ID conflicts or selects a channel with the fewest active networks.

Essentially, the ParseBeaconNotifyIndication allows the device to ignore a beacon if there is a protocol ID or stack profile conflict. This parse-beacon indicator also confirms end device or router capacity.

Additional filters come into play as the device processes the request. For example, a device may check to see if there is space in the neighbor table to save information sent in the response. The following functions provide further filtering for the receiving device.

8.1.3.2 Parent to Join

The function SearchForSuitableParentToJoin selects a potential parent to join from a list formed with the responses sent by the devices that heard the beacon request.

8.1.3.3 Select PAN ID

The function SelectPanId chooses a PAN ID for the device seeking to form a network, when the upper layer specifies NULL as PanId (0xFFFF). This selection is based on the extended PAN ID, the NWK PAN ID, and the link quality, depth, and permit join flags.

8.1.3.4 Select Logical Channel

The MAC layer sends an active scan confirmation invoking the function SelectLogicalChannel. The ZC selects a logical network, with the channel selection criteria set for first one with zero networks, or the one containing the smallest number of PANs.

8.1.4 NWK Layer Interfaces

The macros in Table 8-2 use the given attributeId to call the relevant function.

Table 8-2. NWK Layer Functions and Attributes

Function	Description
NlmeGetRequest(attributeId#)	Get a simple attribute from the NIB (e.g., nwkShortAddress).
NlmeGetRequestTableEntry(attributeId,index)	Get an entry from NIB table attribute (for example, address map)
NlmeSetRequest(attributeId, pValue)	Set a simple attribute from the NIB (for example, nwkShortAddress).
NlmeSetRequestTableEntry(attributeId,index,pValue)	Set an entry from an NIB table attribute (e.g., address map)
IsLocalDeviceTypeARouter()	Returns true or false response (device is or is not a router)
IsLocalDeviceReceiverOnWhenIdle()	Returns true or false response (true = radio always on even if idle)

8.1.5 NWK Layer Filters

These NWK layer filters allow putting in place limits to the networks a given device can hear.

8.1.5.1 Hear Short Address

This function checks for a specific address listed in the *IcanHearYouTable*.

Macro

```
bool_t CanIHearThisShortAddress(uint8_t *pSourceAddress);
```

Returns

- False, if gICanHearYouCounter is anything but 0 and the address given was not in the table
- True, otherwise

8.1.5.2 Set Table List

This macro sets the IcanHearYouTable.

```
Bool_t SetICanHearYouTable(uint8_t addressCounter, zbNwkAddr_t *pAddressList);
```

Returns

- False, if addressCounter is larger than the IcanHearYouTable.
- True, sets the device list in the table

8.1.5.3 Get Table List

This macro gets a pointer to the destination buffer where the table is going to be copied, along with the size of destination buffer.

Macro

```
index_t GetICanHearYouTable(zbNwkAddr_t *pDstTable, index_t maxElementsInDstTable);
```

Returns

Number of table entries copied to destination buffer and the table list.

8.2 NWK Information Base

The NWK information base (NIB) contains all of the attributes used by the NWK layer when communicating with adjacent layers.

Table 8-3. NWK Information Base Attributes

Attribute	ID	Type	Range	Description	Default
nwkSequenceNumber	0x81	Integer	0x00 - 0xff	Sequence number used to identify outgoing frames	Random value from within range
nwkPassiveAckTimeout	0x82	Integer	0x00 - 0x0a	Maximum time duration in seconds allowed for parent and all child devices to retransmit a broadcast message (passive ACK time-out)	0x03
nwkMaxBroadcastRetries	0x83	Integer	0x00 - 0x5	Maximum number of retries allowed after a broadcast transmission failure	0x03
nwkMaxChildren	0x84	Integer	0x00 - 0xff	The number of children a device is allowed to have on its current network	0x07
nwkMaxDepth	0x85	Integer	0x01 - nwkcMax Depth	Depth a device can have: maximum hops from ZC	0x05
nwkMaxRouters	0x86	Integer	0x01-0xff	Max number of routers any one device is allowed to have as children; This value is determined by the ZC for all devices in the network	0x05
nwkNeighborTable	0x87	Set	Variable	Current set of neighbor table entries in the device	Null set
nwkNetworkBroadcastDeliveryTimel	0x88	Integer	(nwkPassiveAc kTimeouT* nwkBroadcast Retries 0x00 – 0xff	Time duration in seconds that a broadcast message needs to encompass the entire network	nwkPassiveAck Timeout * nwkBroadcastR etries
nwkReportConstantCost	0x89	Integer	0x00-0x01	If set to 0, the NWK layer calculates link cost from all neighbor nodes using LQI values reported by the MAC layer; it reports a constant value otherwise	0x00
nwkRouteDiscoveryRetries Permitted	0x8a	Integer	0x00-x03	Number of retries allowed after an unsuccessful route request	nwkcDiscovery RetryLimit
nwkRouteTable	0x8b	Set	Variable	Current set of routing table entries in the device	Null set

Chapter 9
Application Support Layer

BeeStack augments the communication capabilities of ZDO with the Application Support Layer (ASL). Although not a true layer, ASL generates application and application-layer commands in a form that ZDP can efficiently process.

ASL uses the SAP handlers to send commands to ZDP. For example, the ASL_NWKAddr_req gets taken over by ZDP for processing (see NWK_addr_req).

See the file `ASL_Interface.c` for a list of all the LCD strings.

The application support layer (ASL) includes all of the utility function prototypes for the BeeKit applications.

9.1 ASL Utility Functions

The function prototypes in `ASL_UserInterface.h` include the following:

- void ASL_InitUserInterface(char *pApplicationName);
- void ASL_DisplayChangeToCurrentMode(uint8_t DeviceMode);
- void ASL_UpdateDevice(zbEndPoint_t ep, SystemEvents_t event);
- void ASL_HandleKeys(key_event_t event);
- void ASL_ChangeUserInterfaceModeTo(UIMode_t DeviceMode);
- void ASL_AppSetLed(LED_t LEDNr,LedState_t state);
- void ASL_LCDWriteString(char *pstr);
- void ASL_DisplayTemperature(int16_t Temperature);

9.2 ASL Data Types

The structure `ASL_DisplayStatus_t` keeps track of the LED states for certain modes in an application.

```
typedef struct ASL_DisplayStatus_Tag{
        uint8_t Leds;
} ASL_DisplayStatus_t;
```

The structure `ASL_SendingNwkData_t` keeps the information for the type of communication between applications. For example,

```
typedef struct ASL_SendingNwkData_tag{
        zbAddrMode_t gAddressMode;
        zbGroupId_t  aGroupId;
        zbSceneId_t aSceneId;
        zbNwkAddr_t  NwkAddrOfIntrest;
}ASL_SendingNwkData_t;
```

Table 9-1 shows the messages used by the application for certain events, which can be re-configured by the developer. See also ASL_Interface.h for a list of all the LCD strings.

Table 9-1. ASL User Interface Messages

String Variable	Default Value
gsASL_ChannelSelect[ ]	“Select channel”
gsASL_Running[]	“Running Device”
gsASL_PermitJoinEnabled[]	“Permit Join (E)”
gsASL_PermitJoinDisabled[]	“Permit Join (D)”
gsASL_Binding[]	“Binding”
gsASL_BindingFail[]	“Binding Fail“
gsASL_BindingSuccess[]	“Binding Success“
gsASL_UnBinding[]	“UnBinding“
gsASL_UnBindingFail[]	“UnBinding Fail“
gsASL_UnBindingSuccess[]	“UnBinding Success“
gsASL_RemoveBind[]	“Remove Binding“
gsASL_ResetNode[]	“ResetNode“
gsASL_IdentifyEnabled[]	“Identify Enabled“
gsASL_IdentifyDisabled[]	“Identify Disabled“
gsASL_Matching[]	“Matching“
gsASL_MatchFound[]	“Match Found“
gsASL_MatchFail[]	“Match Fail“
gsASL_MatchNotFound[]"	“No Match Found“

9.3 ASL Utility Functions

The application support library (ASL) includes all of the utility function prototypes for the BeeKit applications.

9.3.1 Initialize User Interface

This function initializes the devices (LEDs and keys), data, and callback functions needed for the user interface.

Prototype

```
void ASL_InitUserInterface(char *pApplicationName);
```

9.3.2 Set Serial LEDs

This function flashes the LEDs in a serial pattern and keeps track of the state of the LEDs when in the application mode.

Prototype

```
void ASL_SerialLeds(void);
```

9.3.3 Stop Serial LEDs

This function stops the serial LEDs flashing and turns off all of them.

Prototype

```
void ASL_StopSerialLeds(void);
```

9.3.4 Set LED State

The function ASL_SetLed sets the state of the LED (LED1, LED2, LED3 or LED4, LED_ALL) to a given state (gLedFlashing_c, gLedStopFlashing_c, gLedOn_c, gLedOff_c, gLedToggle_c) in the application mode, keeping track of all the states changes in this mode.

Prototype

```
void ASL_AppSetLed(LED_t LEDNr, uint8_t state);
```

9.3.5 Write to LCD

This function writes a given string (pstr) on line one of the LCD, when the LCD is supported.

Prototype

```
void ASL_LCDWriteString(char *pstr);
```

9.3.6 Change User Interface Mode

This function indicates to the device the mode in which it is running, and sends as a parameter the mode to change to, either gConfigureMode_c for configuration mode or gApplicationMode_c for application mode.

Prototype

```
void ASL_ChangeUserInterfaceModeTo(uint8_t DeviceMode);
```

9.3.7 Display Current User Interface Mode

ASL_DisplayChangeToCurrentMode displays the device mode when changing between user interface modes. The function uses the parameters gConfigureMode_c for configuration mode or gApplicationMode_c for application mode.

Prototype

```
void ASL_DisplayChangeToCurrentMode(uint8_t DeviceMode);
```

9.3.8 Update Device

Based on the application event, the function `ASL_UpdateDevice` will call certain functions. This function contains all the events common to all applications using the files `ASL_UserInterface.h` and `ASL_UserInterface.c`. Those common events include End Device Bind, Change Mode, toggle identify mode, add group, store scene, and recall scene.

Prototype

```
void ASL_UpdateDevice(zbEndPoint_t ep, uint8_t event);
```

9.3.9 Handle Keys

This function handles the common keys to all applications using the files `ASL_UserInterface.h` and `ASL_UserInterface.c`, regardless of mode (application or configuration).

Prototype

```
void ASL_HandleKeys(key_event_t);
```

9.3.10 Display Temperature

This function displays a temperature value (negative or positive) on the LCD in the form “TEMP = 452 C”.

Prototype

```
void ASL_DisplayTemperature(int16_t Temperature);
```

Chapter 10
BeeStack Common Functions

The BeeStack common prototypes provide helper functions to all layers in BeeStack. These primitives allow, among many things, switching from over-the-air (OTA) to native format (that is, from little-endian to big-endian multi-byte values), as well as specifying the number of elements in an array.

10.1 BeeStack Common Prototypes

The prototypes common to all BeeStack layers include functions that convert to and from native formats to over-the-air formats, as shown in Table 10-1.

Table 10-1. Common Prototypes

Prototype	Description
uint16_t OTA2Native16(uint16_t);	For converting from over-the-air to native format
uint16_t Native2OTA16(uint16_t);	For converting from native to over-the-air format
uint16_t Swap2Bytes(uint16_t);	Used to convert between over-the-air (OTA) format (little endian) and native format (big endian on HCS08).

BeeStack common macros, shown in Table 10-2, include functions that define the member offset and number of elements in arrays.

Table 10-2. Common Macros

Macro	Description
NumberOfElements(array)	Number of elements in an array
MbrOfs(type, member)	Offset of a member within a structure
MbrSizeof(type, member)	Size of a member in a structure
UintOf(p2Bytes) (*(uint16_t *) (p2Bytes))	Casts value to uint16 value

10.2 Common Network Functions

The BeeStack common network functions in Table 10-3 include confirming the NWK address, verifying the NWK address, and copying bytes to overwrite table entries.

Table 10-3. BeeStack Common Network Functions

Functions	Description
bool_t IsSelfIeeeAddress(zbIeeeAddr_t aIeeeAddr);	Is this the node's own IEEE address (True or False)
bool_t IsSelfNwkAddress(zbNwkAddr_t aNwkAddr);	Is this the node's own NWK address?
bool_t IsBroadcastAddress(zbNwkAddr_t aNwkAddr);	Is this one of the broadcast addresses?
bool_t IsValidNwkUnicastAddr(zbNwkAddr_t aNwkAddr);	Is this a valid NWK addr for unicasting?
bool_t IsValidNwkAddr(zbNwkAddr_t aNwkAddr);	Confirm valid NWK address
void BeeUtilLargeMemSet(void *pBuffer, uint8_t value, uint16_t iCount);	Set a large array of memory (larger than FLib_memset can handle
void Copy8Bytes(zbIeeeAddr_t aIeeeAddr1,zbIeeeAddr_t aIeeeAddr2	Copies 8 bytes from one location to another. Assumes they do not overlap. Used throughout the code
bool_t IsEqual8Bytes(zbIeeeAddr_t aIeeeAddr1, zbIeeeAddr_t aIeeeAddr2);	Are the two IEEE addresses equal?
void Fill8BytesToZero(zbIeeeAddr_t aIeeeAddr1);	Fill IEEE (long) address with 0s
void FillWithZero(void *pBuffer, uint8_t size);	Fill any length buffer with 0s
bool_t Cmp8BytesToZero(zbIeeeAddr_t aIeeeAddr1);	Is this IEEE address all 0s?
bool_t Cmp8BytesToFs(zbIeeeAddr_t aIeeeAddr1);	Compare this IEEE address to all 0xFFs
uint16_t Swap2Bytes(uint16_t iOldValue);	Swaps bytes to convert between OTA and native format for a 16-bit word
void Swap2BytesArray(uint8_t *pArray);	Swaps bytes to convert between OTA and native format for a 2-byte array
uint8_t *FLib_MemChr(uint8_t *pArray, uint8_t iValue, uint8_t iLen);	Look for a byte in an array of bytes
void BeeUtilSetIndexedBit(uint8_t *pBitArray, index_t iBit);	Set an indexed bit (used by APS group functions)
uint8_t BeeUtilClearIndexedBit(uint8_t *pBitArray, index_t iBit);	Clear the bit
bool_t BeeUtilGetIndexedBit(uint8_t *pBitArray, index_t iBit);	Get the bit
bool_t BeeUtilArrayIsFilledWith(uint8_t *pArray, uint8_t value, index_t iLen);	Check to see if an array is filled with a particular value

Chapter 11
User-Configurable BeeStack Options

BeeStack gets most of its configuration from BeeKit properties (which are compile-time options). This section explains some of these compile-time options available to the user.

Freescale recommends setting all options through BeeKit.

11.1 Compile-Time Options

These compile-time options can be changed by the user. The compile-time options are included in the `ApplicationConf.h` file.

Table 11-1. ApplicationConf.h Compile-Time Options

Option	Description
mDefaultValueOfChannel_c	Select the default channel(s) on which to form or join the network. Bit mask of channels. Use 0x07fff800 to allow any of the 16 channels (11-26) to form or join a network.
mDefaultValueOfPanId_c	Default value of PAN ID on which to form or join the network. Use 0xffff to choose random PAN ID on which to form, or any PAN ID on which to join.
mDefaultNwkExtendedPANID_c	Default value of extended PAN ID. Use 0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00 to choose the IEEE address of the ZC when forming a network or to mean any extended PAN ID (use mDefaultValueOfPanId_c) when joining a network. Extended PAN ID takes precedence over the PAN ID option.
mDefaultValueOfExtendedAddress_c	The MAC (or IEEE) address of the node. Use 0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00 to request the stack pick a random MAC address; do not use this for production nodes.
mDefaultValueOfAuthenticationPollTimeOut_c	The time (in milliseconds) for a node to poll for joining a network.
mDefaultValueOfPollTimeOut_c	The time (in milliseconds) for and end-device to poll its parent.
gRxOnWhenIdle_d	Set to 0x01 (TRUE) to enable a ZED to continuously power it's receiver. Doesn't use polling.
mDefaultValueOfNetworkKey_c	The default network key. Can be any 128-bit value.
mDefaultValueOfNwkKeyPreconfigured_c	Choose whether a preconfigured key or non-preconfigured key is used. A non-preconfigured key is sent over-the-air in the clear on the last hop when a node joins the network, which is used in a Home Automation network. A preconfigured key must be entered into the node out of band (through a serial port or other application defined method)
gAllowNonSecure_d	Allow non-secure packets to be sent/received on a secure network. Note: this creates a security loop-hole if enabled. Disabled (set to FALSE) by default.
mDefaultValueOfEndDeviceBindTimeOut_c	The time (in milliseconds) for an end-device-bind to timeout on the ZC.

Table 11-1. ApplicationConf.h Compile-Time Options (continued)

mDefaultValueOfPermitJoinDuration_c	Value of permit join. Default is 0xff which is always on. Set to 0x00 to have permit join off when the node starts up.
mDefaultValueOfNwkScanAttempts_c	Number of active scans to request beacons. Set higher if network is dense (many nodes in the vicinity).
mDefaultValueOfNwkTimeBwnScans_c	Time in milliseconds between beacon scans.
mDefaultValueOfAuthTimeOutDuration_c	Timeout for authentication process. Defaults to 0x1188 (5000) milliseconds.
mDefaultReuseAddressPolicy_c	When a child leaves, is it OK to reuse the address? Set to FALSE by default.
gMaxFailureCounter_c	Determines # of times a polling child must fail to contact its parent before trying to rejoin the network.
mDefaultValueOfNwkFormationAttempts_c	This determines how many times to attempt to form the network. This option is ignored.
mDefaultValueOfEndDeviceBindTimeOut_c	Default timeout (in milliseconds) between the ZC receiving one end-device-bind request and the second end-device-bind request. Defaults to 10 seconds.
mDefaultValueOfDiscoveryAttemptsTimeOut_c	Timeout between network discovery attempts. Defaults to 0.
mDefaultValueOfNwkDiscoveryAttempts_c	Number of attempts to discover a network to join. Defaults to 0, which means forever.
mDefaultValueOfBatteryLifeExtension_c	Does this node operate on batteries?
mDefaultValueOfCurrPowerSourceAndLevel_c	Set the current power source and level.
mDefaultValueOfNwkOrphanScanAttempts_c	How many times to attempt to join when orphaned? End devices only. Defaults to 0, which means forever.
mDefaultValueOfNwkSecurityLevel_c	This must always be set to 5.
mDefaultValueOfLpmStatus_c	Enable low power during startup or wait until application enables low power.
gPowerSource_d	Set to TRUE if on batteries.

11.2 More Compile-time Options

Compile-time options available to users in BeeStackConfiguration.h include those macros listed in Table 11-2.

Table 11-2. BeeStackConfiguration.h Compile-Time Options

Option	Description
gNwkInfobaseMaxBroadcastRetries_c	Number of retries on each broadcast. Default is 2.
gCoordinatorNwkInfobaseMaxNeighborTableEntry_c	Number of neighbor table entries on a ZC. Default is 24.
gRouterNwkInfobaseMaxNeighborTableEntry_c	Number of neighbor table entries on a ZR. Default is 25.
gEndDevNwkInfobaseMaxNeighborTableEntry_c	Number of neighbor table entries on a ZED. Default is 6.
gNwkInfobaseMaxRouteTableEntry_c	Number of entries in the routing table. Default is 6.
gNwkMaximumChildren_c	Maximum number of total children (routers + end-devices). Default is 20 and must be 20 for Stack Profile 0x01. Advanced option.
gNwkMaximumRouters_c	Maximum number of routers. Default is 6. Advanced option.
gNwkMaximumDepth_c	Maximum depth from ZC in a tree/mesh network. Default is 5. Advanced option.
gICanHearYouTableCapability_d	Set to 0x01 (TRUE) to enable the I-can-hear-you-table. Allows for easy capture of ZigBee routing behavior by defining which nodes can hear which other nodes.
gDefaultValueOfMaxEntriesForICanHearYouTable_c	Number of entries in the table. Only enabled if gICanHearYouTableCapability_d is enabled.
gApsMaxAddrMapEntries_c	Number of address map entries. Used for binding tables. Default is 5. Set to at least gMaximumApsBindingTableEntries_c.
gMaximumApsBindingTableEntries_c	Number of local binding table entries. Default 5.
gApsMaxGroups_c	Number of local group table entries. Default 5.
gApsMaxRetries_c	Maximum # of retries by APS layer. Default 3.
gApsAckWaitDuration_c	Wait (in milliseconds) between APS retries. Default is 1800 milliseconds or 1.8 seconds.
gScanDuration_c	The scan duration for energy detect and active scans, as defined by the ZigBee specification (an exponential scale).
gNWK_addr_req_d	Enable request.
gNWK_addr_rsp_d	Enable response.
gIEEE_addr_req_d	Enable request.
gIEEE_addr_rsp_d	Enable response.
gNode_Desc_req_d	Enable request.
gNode_Desc_rsp_d	Enable response.
gPower_Desc_req_d	Enable request.
gPower_Desc_rsp_d	Enable response.

Table 11-2. BeeStackConfiguration.h Compile-Time Options

gSimple_Desc_req_d	Enable request.
gSimple_Desc_rsp_d	Enable response.
gActive_EP_req_d	Enable request.
gActive_EP_rsp_d	Enable response.
gMatch_Desc_req_d	Enable request.
gMatch_Desc_rsp_d	Enable response.
gComplex_Desc_req_d	Enable request.
gComplex_Desc_rsp_d	Enable response.
gUser_Desc_req_d	Enable request.
gUser_Desc_rsp_d	Enable response.
gDiscovery_Cache_req_d	Enable request.
gDiscovery_Cache_rsp_d	Enable response.
gEnd_Device_annce_d	Enable end-device-announce when a node joins or rejoins the network.
gUser_Desc_set_d	Enable request.
gUser_Desc_conf_d	Enable response.
gSystem_Server_Discovery_req_d	Enable request.
gSystem_Server_Discovery_rsp_d	Enable response.
gDiscovery_store_req_d	Enable request.
gDiscovery_store_rsp_d	Enable response.
gNode_Desc_store_req_d	Enable request.
gNode_Desc_store_rsp_d	Enable response.
gPower_Desc_store_req_d	Enable request.
gPower_Desc_store_rsp_d	Enable response.
gActive_EP_store_req_d	Enable request.
gActive_EP_store_rsp_d	Enable response.
gSimple_Desc_store_req_d	Enable request.
gSimple_Desc_store_rsp_d	Enable response.
gRemove_node_cache_req_d	Enable request.
gRemove_node_cache_rsp_d	Enable response.
gFind_node_cache_req_d	Enable request.
gFind_node_cache_rsp_d	Enable response.
gEnd_Device_Bind_req_d	Enable request.
gEnd_Device_Bind_rsp_d	Enable response.

Table 11-2. BeeStackConfiguration.h Compile-Time Options

gBind_req_d	Enable request.
gBind_rsp_d	Enable response.
gUnbind_req_d	Enable request.
gUnbind_rsp_d	Enable response.
gBind_Register_req_d	Enable request.
gBind_Register_rsp_d	Enable response.
gReplace_Device_req_d	Enable request.
gReplace_Device_rsp_d	Enable response.
gStore_Bkup_Bind_Entry_req_d	Enable request.
gStore_Bkup_Bind_Entry_rsp_d	Enable response.
gRemove_Bkup_Bind_Entry_req_d	Enable request.
gRemove_Bkup_Bind_Entry_rsp_d	Enable response.
gBackup_Bind_Table_req_d	Enable request.
gBackup_Bind_Table_rsp_d	Enable response.
gRecover_Bind_Table_req_d	Enable request.
gRecover_Bind_Table_rsp_d	Enable response.
gBackup_Source_Bind_req_d	Enable request.
gBackup_Source_Bind_rsp_d	Enable response.
gRecover_Source_Bind_req_d	Enable request.
gRecover_Source_Bind_rsp_d	Enable response.
gMgmt_NWK_Disc_req_d	Enable request.
gMgmt_NWK_Disc_rsp_d	Enable response.
gMgmt_Lqi_req_d	Enable request.
gMgmt_Lqi_rsp_d	Enable response.
gMgmt_Rtg_req_d	Enable request.
gMgmt_Rtg_rsp_d	Enable response.
gMgmt_Bind_req_d	Enable request.
gMgmt_Bind_rsp_d	Enable response.
gMgmt_Leave_req_d	Enable request.
gMgmt_Leave_rsp_d	Enable response.
gMgmt_Direct_Join_req_d	Enable request.
gMgmt_Direct_Join_rsp_d	Enable response.
gMgmt_Permit_Joining_req_d	Enable request.
gMgmt_Permit_Joining_rsp_d	Enable response.

Table 11-2. BeeStackConfiguration.h Compile-Time Options

gMgmt_Cache_req_d	Enable request.
gMgmt_Cache_rsp_d	Enable response.
gSystemEventEnabled_d	Tell application about ZDO and system events. Defaults to TRUE.
gNumberOfEndPoints_c	Maximum number of application endpoints supported by the node. Default is 5.

Chapter 12
BeeStack Security

BeeStack supports full ZigBee security for stack profile 0x01 of the ZigBee 2006 specification.

12.1 Security Overview

Adding security into a ZigBee network has the following effects

- Every data packet (at the network payload level) is encrypted with AES 128-bit encryption. This means that 802.15.4 radios not on the ZigBee network will not be able to understand the packets sent over-the-air.
- Every packet is authenticated using the same AES 128-bit encryption engine and a 32-bit frame counter. This means that 802.15.4 radios not on the ZigBee network will not be able to send over-the-air data to any node in the network, even using a direct replay of the octets from a previous message.
- Over-the-air packets grow by 15 bytes, with the addition of the AUX header. See Figure 12-1 below.
- Transmitting becomes slightly slower (by about 5ms per encode/decode).

While the network is protected from replay attacks, ZigBee security does not prevent

- Denial of service attacks. Any 802.15.4 radio could be put into constant transmit mode using up all bandwidth in the local vicinity.
- Rogue nodes from hearing the key when a node joins a network (only if non-preconfigured key is used. Preconfigured keys are never transmitted in the clear).

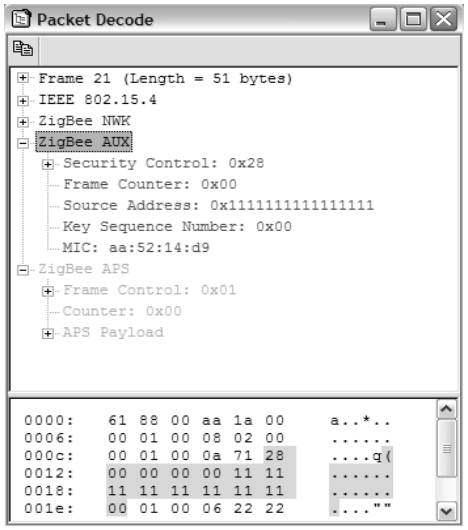

Figure 12-1. Secure ZigBee Packet Example

ZigBee security shares a network key among all nodes in the network, sometimes called a symmetric key. It is assumed in ZigBee that the network is generally closed (joining disabled) and that if a node is allowed on the network, that the node is trusted. The ZigBee trust center (on the ZigBee Coordinator) has the ability to kick nodes off the network, or deny them access in the first place.

ZigBee security comes in two modes:

Preconfigured key — Means that each node somehow “knows” the network key out-of-band, perhaps installed at the factory or by a special commissioning tool. The key is never sent over the air and can be securely updated to a new key.

Non-preconfigured key — Used in less secure networks, such as home automation. The key is sent (last hop only) in the clear.

In addition to the standard ZigBee security, BeeStack offers the ability to send unsecured packets in a secure network. This behavior is not compatible with the ZigBee public profiles standard but could be used in private profiles. To send unsecured packets on a secure network, disable the gApsTxOptionSecEnabled_c field in the txOptions flags of the AF_DataRequest().

12.2 Security Configuration Properties

The following security properties can be modified in BeeKit to configure BeeStack security.

12.2.1 mDefaultValueOfNwkKeyPreconfigured_c

Set mDefaultValueOfNwkKeyPreconfigured_c to 1 to enable a preconfigured key (key obtained out-of-band). Set it to 0 to enable non-preconfigured key (over-the-air key transport).

12.2.2 mDefaultValueOfNwkSecurityLevel_c

Always use mDefaultValueOfNwkSecurityLevel_c level 5 for compatibility with ZigBee stack profile 0x01. The other ZigBee security levels are listed below for completeness.

12.2.3 mDefaultValueOfNetworkKey_c

The mDefaultValueOfNetworkKey_c property lists the key that will be used by BeeStack as the initial key. This key as a 128-bit key (for use with AES 128-bit encryption), and can be any value other than 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 (all zeros).

12.2.4 gDefaultValueOfMaxEntriesForExclusionTable_c

The gDefaultValueOfMaxEntriesForExclusionTable_c property determines how many entries are in the exclusion table. These will be automatically excluded from joining the network by the trust center in a secure network. See the next section for more details.

12.3 ZigBee Trust Center Authentication

The trust center resides on the ZigBee Coordinator. This trust center is the only device that allows a node on a secure ZigBee network. The trust center is not required for normal operation, only for adding nodes or switching the security key.

The trust center has the opportunity to disallow nodes from joining the network. BeeStack has a built-in exclusion table, or the application can modify the function

```
bool_t deviceInExclusionTable(uint8_t *pIeeeAddress)
```

to include any sort of algorithm to include or exclude a node. The ZigBee specification only provides the IEEE (sometimes called MAC) address for this purpose. There is no other data about the node wishing to join the network. This function can be found in `ZdoNwkManager.c`.

A node can be forced off the network. To do this, use prototype for the leave request. See ZDP for a complete discussion of this ASL interface to ZDP.

```
void ASL_Mgmt_Leave_req
(
        zbCounter_t  *pSequenceNumber,
        zbNwkAddr_t  aDestAddress,
        zbIeeeAddr_t aDeviceAddres,
        zbMgmtOptions_t  mgmtOptions
);
```