Freescale BeeStack™

Software Reference Manual for ZigBee 2007

ARM POWERED®

freescale™
semiconductor

Contents

Chapter 1
Introduction

Chapter 2
ZigBee Overview

Chapter 3

BeeStack Features

Chapter 4
Application Framework

Chapter 5
Application Support Sub-layer

BeeStack™ Software Reference Manual for ZigBee 2007, Rev. 1.1

Chapter 6
ZigBee Device Objects

Chapter 7
ZigBee Device Profile

Chapter 8
Network Layer

Chapter 9
Application Support Layer

Chapter 10
BeeStack Common Functions

Chapter 11
User-Configurable BeeStack Options

Chapter 12
BeeStack Security

BeeStack™ Software Reference Manual for ZigBee 2007, Rev. 1.1

Chapter 13
Permission Configuration Table

Chapter 14
Frequency Agility

Chapter 15

BeeStack™ Software Reference Manual for ZigBee 2007, Rev. 1.1

vi   Freescale Semiconductor

Interpan Communication

Appendix A
Porting from BeeStack 2006 to BeeStack 2007

Appendix B
Table and Buffer Sizes

About This Book

This manual describes BeeStack, the Freescale implementation of the ZigBee wireless network protocol stack for the ZigBee 2007 specification. 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 and stack profile 0x02 of the ZigBee 2007 specification.

Chapter 13   Permission Configuration Table - describes the optional BeeStack Permission Configuration Table feature.

Chapter 14   Frequency Agility – describes how BeeStack supports an example implementation of a frequency agility channel master which demonstrates how frequency agility could be implemented.

BeeStack™ Software Reference Manual for ZigBee 2007, Rev. 1.1

Chapter 14   Interpan Communication – details how BeeStack supports the Inter pan
                    communication method that is specified by the Smart Energy/AMI application
                    profile specification. The Interpan communication feature allows for
                    communication outside the ZigBee network to very simple devices.

Appendix A   Porting ZigBee 2006 to ZigBee 2007 – describes how to port from ZigBee 2006
                    to ZigBee 2007.

Revision History

The following table summarizes revisions to this document since the previous release (Rev. 1.0).

Revision History

Location	Revision
Entire document	Updated for MC1322x silicon revision and numerous software updates.

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

BeeStack™ Software Reference Manual for ZigBee 2007, Rev. 1.1

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
ZigBee Feature Set	Stack Profile 0x01
ZigBee Pro Feature Set)	Stack Profile 0x02
ZDO	ZigBee device object(s)
ZDP	ZigBee device profile
ZED	ZigBee End Device
ZC	ZigBee Coordinator
ZR	ZigBee Router
Zx	Freescale Specific Combo Device
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 053474r17, *ZigBee Specification*, ZigBee Alliance, October 2007
2. Document 075123r02, *AFG-ZigBee_ClusterLibrary_Specification*, ZigBee Alliance, May 2008
3. Document 053520r25, *ZB_HA_PTG-Home-Automation-Profile*, ZigBee Alliance, October 2007

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.

BeeStack supports the following ZigBee 2007 specification profiles:

- For the HCS08 Platform
  - — Stack profile 0x01(ZigBee Feature Set)
- For the MC1322x Platform
  - — Stack Profile 0x01 (ZigBee Feature Set)
  - — Stack Profile 0x02 (ZigBee Pro Feature Set)

See Appendix A, "Porting from BeeStack 2006 to BeeStack 2007" for information on how to port from ZigBee 2006 to ZigBee 2007.

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 for ZigBee 2007*, (ZAUGZB2007)
- *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.
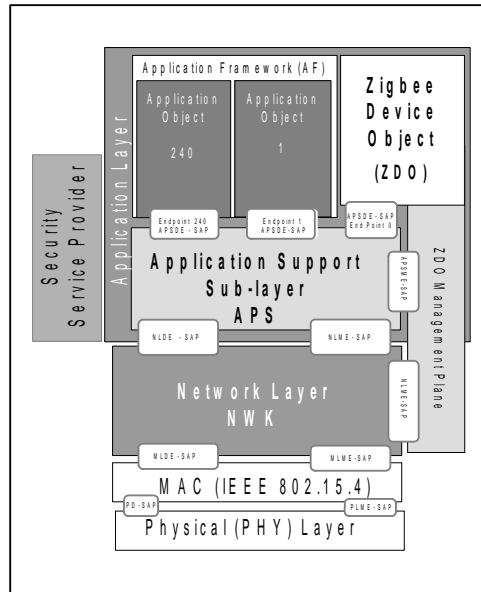
Figure 2-1. ZigBee Layers

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　Combo Device

BeeStack offers multiple library configurations for each device type, including a Combo device (Zx) type. The Combo device type is not a ZigBee defined type, but a Freescale specific stack configuration that allows the application to perform a runtime selection of the ZigBee device types (ZC, ZR, ZED). This allows for creating advanced devices that, for example, can first attempt to join a network as a Router, but if none are found, can restart itself as a Coordinator and form a network. Using the Combo device increases BeeStack's code size footprint compared to selecting a specific device type because the Combo device contains the functionality of all the device types.

2.1.1.5　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.
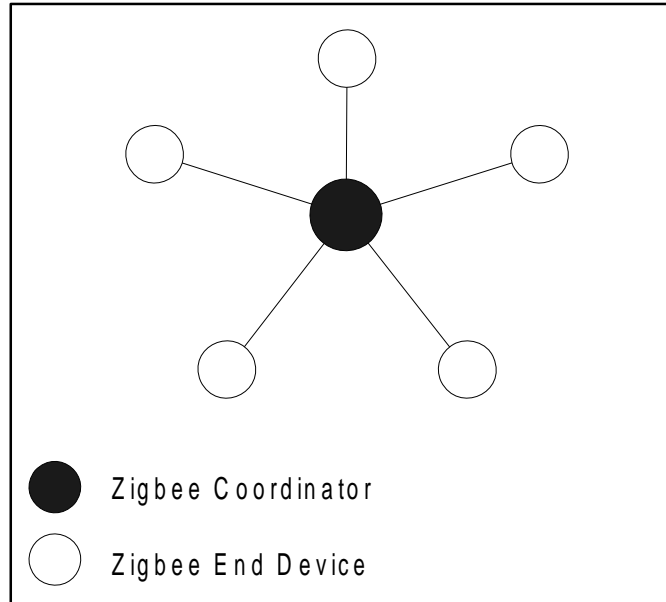
Figure 2-2. Standard Star Network Configuration

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.
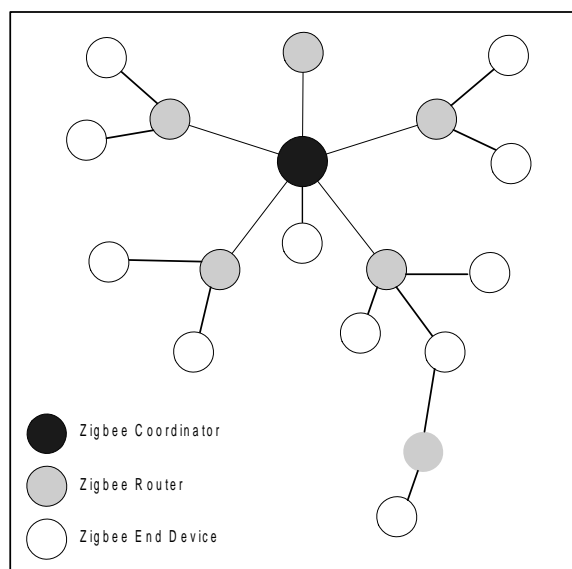
Figure 2-3. Typical Tree Network

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.
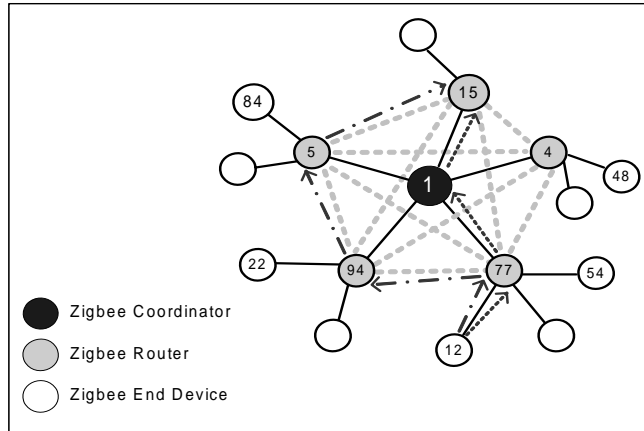
Figure 2-4. Mesh Network Configuration

2.2   ZigBee Feature Sets, Stack Profiles and Application Profiles

Both the 802.15.4 MAC/PHY and ZigBee specification specify many features and functionality in one common specification, but they do not specifically dictate what is actually implemented. This is defined in a stack profile also called a "Feature Set". On top of the ZigBee specification resides the Application profiles which specify the applications and may restrict or make features normally optional in the stack mandatory. It is important that ZigBee developers become familiar with all of these specifications to better understand what features and functionality are required to create a product that can be certified. Figure 2-5 shows how the ZigBee specifications interrelate.
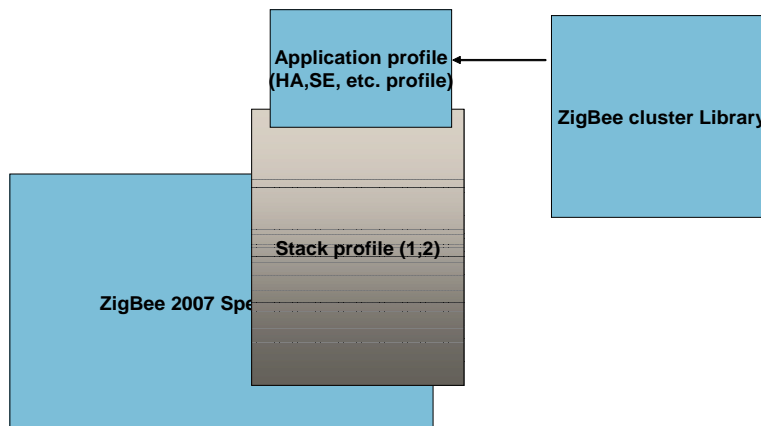
Figure 2-5. Specification Interrelation

The stack profile defines the stack settings, knob and features. All devices in a network must conform to the same stack profile to ensure a working and interoperable network. The ZigBee Alliance has defined two feature sets:

- Stack Profile 0x01 (ZigBee Feature Set)
- Stack Profile 0x02 (ZigBee Pro Feature Set)

The following sections describe the differences between these two feature sets.

2.2.1   Stack Profile 0x01 (ZigBee Feature Set)

- Supports Ad-hoc self forming networks
- Mesh, Tree and Star Networks
- Non-Beaconed only
- Supports 802.15.4 Device Types
- FFD Device (Coordinator and Router)
- Reduced Function Device (End Device)
- Application Support
- Device and Service Discovery
- Messaging with optional responses
- Public and private profiles
- Profiles are predefined Stack and Application attributes
- Security
- Symmetric Key with AES-128
- Authentication and Encryption at NWK and Application levels
- Network Keys and Link Keys (network keys are mandatory)
- Qualification
- Platform and Network Compliance Certification
- Interoperability Events
- Fragmentation (Optional) – allows data that is bigger than a single ZigBee package to be fragmented, sent in multiple packets and re-assembled
  - — Pros – Provides a standard way to handle larger data transfers
- Frequency Agility (Mandatory) – provides a method for the network to change channels in the event of interference
  - — Pros - Provides an ability to detect potential interference on a channel and direct devices on the network to change to a better channel
- PAN ID Conflict Resolution (Mandatory) – provides the ability for separate PANs (networks) to resolve a conflict in the PAN ID without have to restart the entire network
  - — Pros – Provides a way for a network to handle other co-located networks without having to restart a network to assign a new PAN ID.

- InterPan Communication (Non ZigBee feature) – provides the ability for communicating to low cost 802.15.4 devices. This feature was originally defined by the Smart Energy profile but can be used for Inter Pan communication outside of the ZigBee network

2.2.2 Stack Profile 0x02 (ZigBee Pro Feature Set)

The ZigBee Pro feature set uses the ZigBee feature set as a baseline. The ZigBee Pro feature set contains the following features either removed or added from the ZigBee feature set as follows:

Removed

- Tree Routing
  - — Pros - Backup routing mechanism to avoid large routing table's being filled
  - — Cons - Does not work when the parent-child link is broken
- CSKIP address assignment
  - — Pros - Assigns a unique address based on the device's parent. It is decentralized and automatically avoids conflicts
  - — Cons - Addresses are predefined taking up memory in parent device and are network location specific (based on where the device is in the tree), creating issues if the device moves or has to associate with another parent.

Added

- Centralized Data Collection and Network Scalability is provided by adding many-to-one routing and Source routing to the specification
  - — Pros - Supports central concentrator data transmission pattern (many devices communicating with a single device) and occupies just a single table route entry to reach the concentrator. This enables the concentrator to send responses back to individual devices without table routing entries for each device
  - — Cons - If there are many concentrators, then a table entry is needed in each device for every concentrator and the header portion of a message is increased which reduces the payload
- Automated Device Address Management is provided by adding stochastic address assignment/address conflict resolution
  - — Pros - Eliminates the need for the parent to store address tables to assign to devices
  - — Cons - Must employ a broadcast message to ensure there are not duplicate addresses on the network and provide a distributed protocol to detect and notify devices of address conflicts
- Group Addressing is provided by adding multicast to the specification and puts limits on the normal broadcast message to reduce the network traffic created. This is an alternative to APS group addressing which is part of the ZigBee feature set
  - — Pros - Enables a single message to reach a group of devices without sending the message to all devices in the network
  - — Cons - The group must be co-located close to each other to see the benefit over a normal broadcast message and devices do not acknowledge that the command was received

- High Security Modes - Enables a higher level of security than the ZigBee and ZigBee Pro standard security that is typically required for financial transactions
  - Pros - Addresses broadcast replay attacks, ensures that all routers authenticate their router neighbors and enables network key to be encrypted using public key, etc.
  - Cons - Can not support devices that use standard security and will require much more complexity and memory
- Link Status/Symmetric routes
  - Pros - Provides the ability to use the same link for forward and backward routing which reduces table entries and ensures that the devices can communicate both ways
  - Cons - Regular communication between every router and each of its neighbors has to occur to update link quality status

2.2.3   Other ZigBee Configurations

Freescale provides a wide range of possible stack configurations. These can be used for manufacturer specific profiles (MSP) and/or proprietary networks. These configurable options allow users to select those features required and gain the code size savings of removing unneeded features which are otherwise mandatory. Through BeeKit, it is possible to select, enable or disable stack features to tailor the project configuration for the desired functionality and free up resources for application use.

If an MSP/Proprietary network is created, the stack profile ID must be set to ZERO. This is done by setting the gMSPstackProfileEnabled_d property to FALSE, and setting gAppStackProfile_c to the desired stack profile ID value. Both are configurable using BeeKit.

2.2.4   Application Profiles

Application profiles specify a set of applications and can apply further restrictions or make optional features mandatory. The application profiles cover areas such as Home Automation, Commercial Building Automation, Smart Energy and Telecom applications. All application profiles utilize the ZigBee Cluster Library Specification. The ZigBee Cluster Library Specification dictates all services and features used by the application profiles. For example, an On/Off Input Service (such as a lamp or a power outlet) or a Demand Response Load Control Service utilized by smart energy devices to control temperature or shut off devices during critical power periods.

2.3   Routing

The ZigBee feature set supports tree and mesh routing. The ZigBee Pro feature set supports mesh, many to one and source routing. The following sections briefly describe how each of these features operate.

2.3.1   Tree Routing

Tree routing is the fallback routing mechanism in the ZigBee Feature Set. If a mesh route can not be found, a packet is routed using the tree routing mechanism. In tree routing, all messages 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.

BeeStack™ Software Reference Manual for ZigBee 2007, Rev. 1.1

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 passes the message to its own parent.

Tree routing depends on the CSKIP addressing scheme used by the ZigBee Feature Set to determine whether to route the packet up or down in the network.

2.3.2 Mesh Routing

ZigBee utilizes the Ad Hoc On-Demand Vector routing protocol for mesh routing. To discover a mesh route, the originator sends out a route request broadcast. When other nodes receive the route request, they determine whether it know the destination and prepare either to forward the route request in the network or prepare a route reply. When the route request is forwarded, the path cost of the received route request increments with the cost of the received route request. When a reply is sent, the reply then contains the total cost of the "path" to the destination. This allows the originator to determine what route has the lowest cost if multiple route replies are received. Parent nodes will respond on behalf of ZEDs. The stack maintains a route discovery table which insures that multiple replies are not sent for the same route request. This table also limits how many route requests can be handled at the same time. The table is configurable by adjusting gNwkRoutingMaxRouteDiscoveyTableEntry_c . Mesh routes are discovered automatically when communicating with a node where the route is not known. It is also possible for the application to force a route reply by setting a Tx Option bit (gApsTxOptionForceRouteDiscovery_c), or calling the NLME-route-discovery request function. Each node also maintains a route table. This table stores resolved routes.

A route age expiry feature is available in BeeStack Codebase 3.0.0 and later. The old route table entry is used if the route table is full. It is possible to modify the behavior of the route expiration algorithm in the ExpireAndGetEntryInRouteTable() function.

2.3.3 Many-to-one and Source Routing

Many-to-one and source routing solves the problem with large networks that require large mesh route tables. In ZigBee networks, there are typically one or more "gateways" or concentrators where many nodes send data. This normally spins off many route discoveries from all of these nodes to establish a communication path to the concentrator and back. Many-to-one and source routing helps reduce the amount of traffic required and reduces route table size.

ZigBee defines a gateway as a "concentrator". The concentrator sends out a many-to-one route request to establish in-bound mesh routes as shown in Figure 2-6. This only adds one route entry in the route tables so nodes can communicate with the concentrator. This greatly reduces the over the air messages required to establish routes from all the devices that want to communicate with the concentrator because normal route discovery is no longer required.
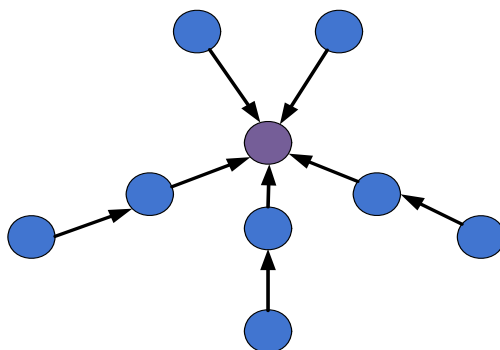
Figure 2-6. Man-to-One Routing

Source routing, as shown in Figure 2-7, allows the concentrator to reply to nodes in the network without establishing new mesh routes. When a node wants to send a packet to a concentrator, it first sends out a route record command. The route record command is relayed from the originating node to the concentrator and it "records" the route. This allows the concentrator to reply back using this "source route".

NOTE

> This reply with a source route takes place if the data concentrator requests a source route during the many-to-one route request. The source routes are stored in the source route table of the concentrator if available.

A concentrator can be memory constrained, which means it does not have a source route table or it can be a very small concentrator. A concentrator that has a large RAM, is able to contain the source routes of numerous nodes. This allows the nodes to send a route record when requested, rather than every time it tries to communicate with a concentrator.
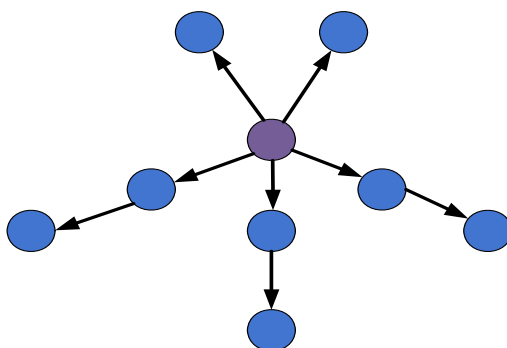
Figure 2-7. Source Routing

To enable the concentrator feature, the gConcentratorFlag_d property must be set to TRUE.

The application can initiate a many to one route discovery by using the ASL_SendRouteDiscoveryManyToOne()function, or configure the stack to send it out with a fixed interval by setting the gConcentratorDiscoveryTime_c property. the radius of the Many to one route request is set by the gConcentratorRadius_c property

The source route table size can be adjust by setting the gNwkInfobaseMaxSourceRouteTableEntry_c property. If the Concentrator should be a non-memory constrained (high ram) Concentrator the property gNwkHighRamConcentrator_d must be set to false.

BeeStack™ Software Reference Manual for ZigBee 2007, Rev. 1.1

All functions for handling the source route table are available in full source code so they can be modified for access for example, external memory (ResetSourceRouteTable(), GetFreeEntryInSourceRouteTable(), NwkRetrieveSourceRoute(), and NwkStoreSourceRoute().

2.4 Groupcast and Multicast

The ZigBee feature set and the ZigBee Pro feature set can address a group of nodes. The ZigBee Feature set supports a normal group transmission where the a network uses the broadcast address and a field in the APS header specifies the group address. The radius of the data request sets the limit for how far the transmission will travel in the network. This means that a typical groupcast is always circular and always a single island as shown in Figure 2-8.
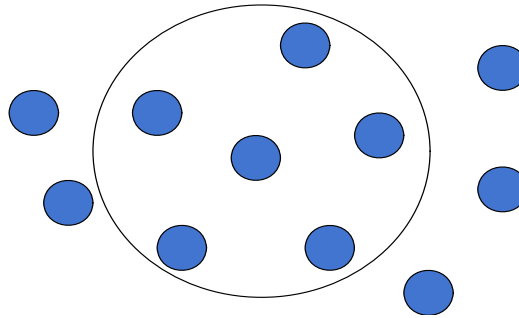
Figure 2-8. Groupcast

The ZigBee Pro feature set also supports a group address scheme called multicast as shown in Figure 2-9. Multicast is enabled if the useMulticast NIB is set to TRUE (default setting) in the mDefaultValueOfNwkUseMulticast_c.

A multicast employs a MAC broadcast and the group address at the network level. This allows the packet to travel within a group and hop to another island of nodes without spamming the entire network with broadcasts. The non member radius AIBsetting (gApsNonMemberRadiusDefault_c) defines how many hops a packet takes outside a group before it "dies".
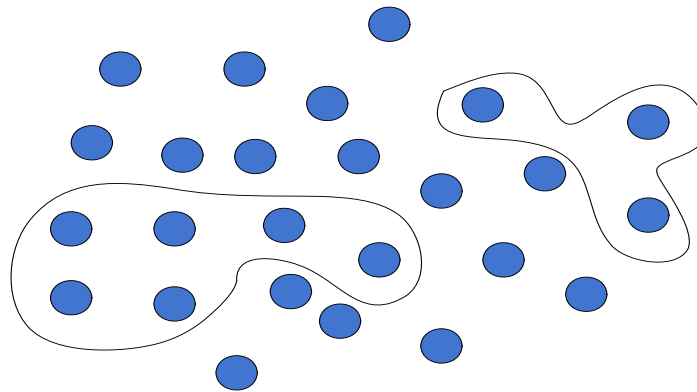
Figure 2-9. Multicast

Both group and multicast schemes look the same from the application perspective. Both use group mode binding or APSDE-DATA.requests. Data is delivered only to endpoints with a matching 16-bit group ID. The only difference is the over the air behavior.

2.5   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, mesh or mesh-tree configurations, as required by the user's application. Stack Profile 0x01(ZigBee Feature Set) uses mesh tree configuration and Stack Profile 0x02 (ZigBee Pro Feature Set) uses mesh only configuration.

2.6   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".

2.7   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".

2.8   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

BeeStack™ Software Reference Manual for ZigBee 2007, Rev. 1.1

      — 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. BeeStack Codebase 3.0.0 or later has a 64 bit address resolution feature where the APS layer attempts to discover the 16 bit address prior to sending the packet.

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.

2.9   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 2-10, 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.
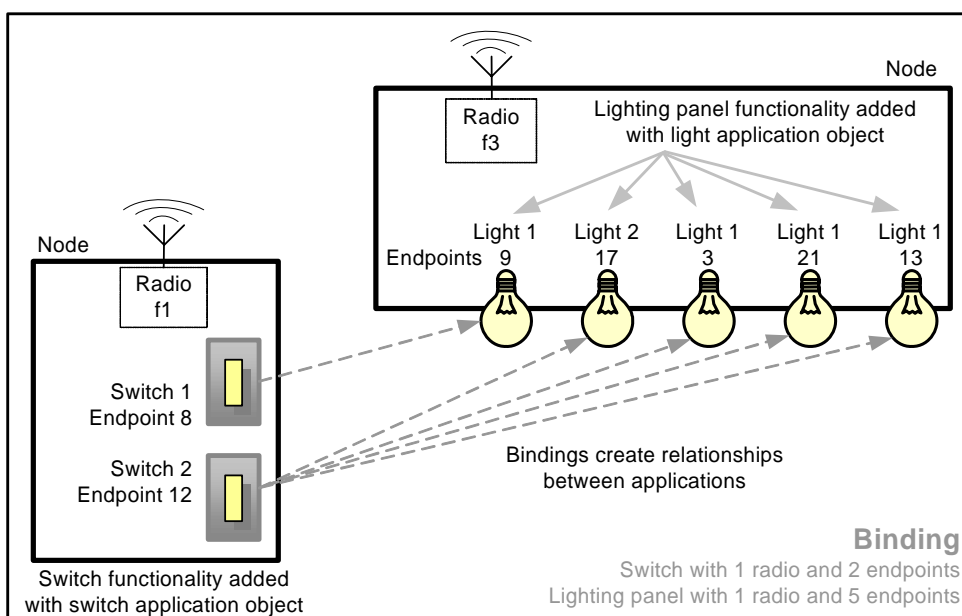
Figure 2-10. Binding and Application Objects

2.10   Application Elements

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

2.10.1   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.

2.10.2   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.

2.10.3   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.
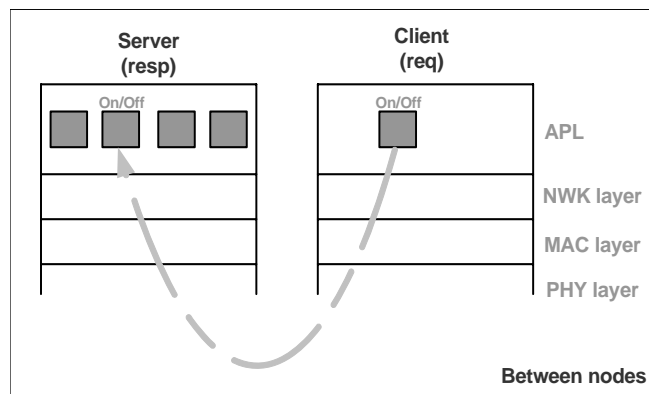
Figure 2-11. On and Off Lighting Application Stack Behavior

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

2.10.4   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.
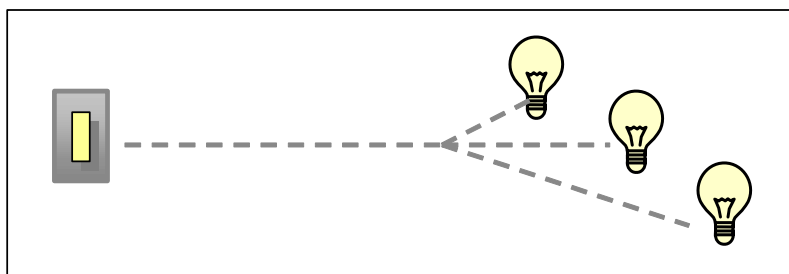
Figure 2-12. Endpoints in ZigBee Network

Endpoints play three major roles in BeeStack. They allow the following:

- 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

BeeStack™ Software Reference Manual for ZigBee 2007, Rev. 1.1

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.
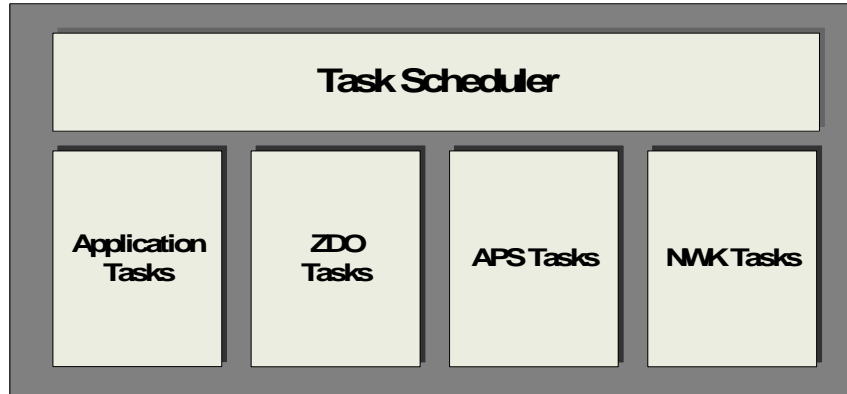
Figure 3-1. Task Scheduler Functionality

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, and frequency agility 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` (for HCS08)
- Binary file format `.bin` (for MC1322x)
- Memory maps: `.map`

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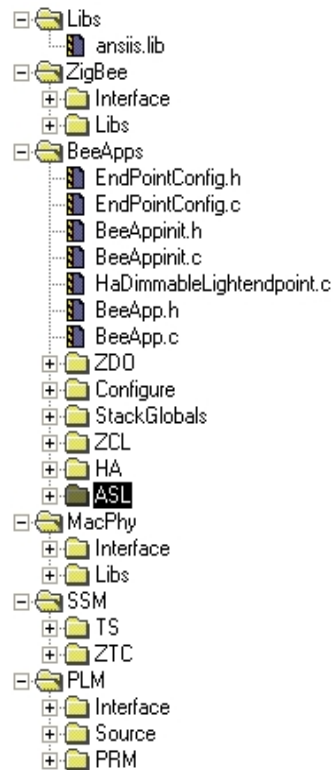
Figure 3-2. Library and Application File Directories

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.
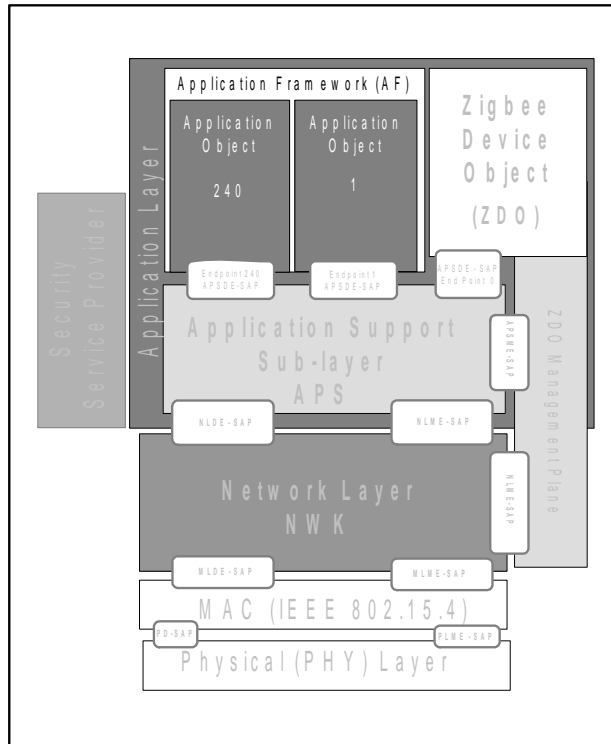
Figure 4-1. BeeStack Layers and Application Framework

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 and Deallocation

`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.

AF_MsgAllocFragment() allows an application to allocate messages for building a larger packet than the regular AF_DataRequestNoCopy() supports. This feature is called fragmentation and the AF_DataRequestFragmentedNoCopy() function must be used. Normally the command AF_DataRequestFragmented() only copies the payload like the `AF_DataRequest()`.

*   AF_FreeDataIndicationMsg() must be used to free a data indication and will take care of freeing all messages if the data indication is a fragmented packet that uses multiple messages.
*   AF_FreeDataRequestFragments() is used for freeing the messages allocated in a fragmentation Data request in the case that not all messages could be allocated to support the total packet payload.

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. If Fragmentation is enabled the number of big buffers should be increased to accommodate the amount of data that the application will send. For every 80 bytes above 78 the number of big buffers should be increased by 1.

NOTE

> The configuration of Big Buffers also depends on whether High Security is used (ZigBee Pro feature set only) because extra storage is needed for entity authentication.

4.3.1   AF_MsgAlloc

Allocate a message buffer for building a packet to be sent via AF_DataRequestNoCopy(). Generally used when building larger payloads in-place. Alternately, use AF_DataRequest(), where it will make a copy of the payload.

Prototype

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

Returns

*   MSG_Alloc(gMaxRxTxDataLength_c);

4.3.2   AF_MsgAllocFragment

Allocates a message buffer for fragmented APSDE-DATA.requests. To be used with the function AF_DataRequestFragmentedNoCopy(). Builds a linked list automatically for the application.

BeeStack™ Software Reference Manual for ZigBee 2007, Rev. 1.1

Prototype

```
void *AF_MsgAllocFragment(void *pHead, uint8_t iFragLen, uint8_t **ppData)
```

Returns

- Allocates a message buffer for fragmented APSDE-DATA.requests
- Builds a linked list automatically for the application

4.3.3   AF_FreeDataIndicationMsg

Free the entire data indication including any messages which used fragmented data.

Prototype

```
void AF_FreeDataIndicationMsg(apsdeToAfMessage_t * pApsdeDataIndMsg)
```

Returns

- None

4.3.4   AF_FreeDataRequestFragments

Frees all fragments in a Data Request. Used if application could not allocate messages enough for the entire message.

Prototype

```
void AF_FreeDataRequestFragments(void *pHead)
```

Returns

- None

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 four functions for sending data, one that copies the application's data payload, AF_DataRequest(), and one that leaves the payload in place, AF_DataRequestNoCopy(). the last two functions are for packets that are larger than the payload that AF_DataRequest supports, AF_DataRequestFragmented(), and one that leaves the payload in place, AF_DataRequestFragmentedNoCopy()

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()` and `AF_DataRequestFragmented()` send 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);

zbStatus_t AF_DataRequestFragmented(afAddrInfo_t *pAddrInfo, uint16_t payloadLen,void
*pPayload, zbApsCounter_t *pConfirmId);
```

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

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

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()` or `AF_DataRequestFragmented()` 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 two no-copy interfaces 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. `AF_DataRequestFragmentedNoCopy()` should be used to avoid allocating very large static buffers in the application, or to copy data directly from e.g. a UART driver to the messages used to send the data request. This will save processing time and RAM.

The prototypes for the no-copy data requests are as follows:

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

zbStatus_t AF_DataRequestFragmentedNoCopy(afAddrInfo_t *pAddrInfo, 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 functions

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

Example using `AF_DataRequestNoCopy()`:

```
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);
}
```

Example using `AF_DataRequestFragmentedNoCopy()`:

```
void Send270BytesNoCopy(afAddrInfo_t *pAddrInfo)
  {
    void *pHead;
    void *pMsg;
    uint8_t *pData;

    // allocate 270 bytes, and fill with 90 x 'A', 'B' and 'C'
    pHead = AF_MsgAllocFragment(NULL, 90, &pData);
    if(pHead)
    {
      FLib_MemSet(pData, 'A', 90);
      pMsg = AF_MsgAllocFragment(pHead, 90, &pData);
    }
    if(pMsg)
    {
      FLib_MemSet(pData, 'B', 90);
      pMsg = AF_MsgAllocFragment(pHead, 90, &pData);
    }
    if(pMsg)
      FLib_MemSet(pData, 'C', 90);

    // send the 270bytes over-the-air
    AF_DataRequestFragmentedNoCopy(pAddrInfo,270,pHead,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 — no APS security (NWK layer only).
- gApsTxOptionSecEnabled_c — APS layer security enabled.
- gApsTxOptionUseNwkKey_c — NWK key used for APS layer security.
- 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 if the receiving node received the packet. ACKs cause more network traffic.
- gApsTxOptionNoSecurity_c — Send a non-secure packet in a secure network. This is a Freescale specific option.

BeeStack™ Software Reference Manual for ZigBee 2007, Rev. 1.1

- gApsTxOptionSuppressRouteDiscovery_c — Normally, packets discover a route if needed. Turn off route discovery to route along the tree. This is a Freescale specific flag option and is only applicable for ZigBee Feature Set.
- gApsTxOptionForceRouteDiscovery_c — Normally, packets discover a route if needed. Turn on force route discovery to discover a route before sending the packet. This is a Freescale specific option.
- gApsTxOptionFragmentationRequested_c — enable Fragmentation Enabled. This bit is automatically set when using the fragmentation data requests. This is a Freescale specific option.

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 Stack Profile 0x01 (ZigBee Feature Set).

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 */
    AF_FreeDataIndicationMsg(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.

BeeStack™ Software Reference Manual for ZigBee 2007, Rev. 1.1

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:

```
ttypedef struct zbApsdeDataIndication_tag
{

  zbAddrMode_t    dstAddrMode;     /* address mode */
  zbNwkAddr_t     aDstAddr;        /* destination address or group */
  zbEndPoint_t    dstEndPoint;     /* destination endpoint */
  zbAddrMode_t    srcAddrMode;     /* address mode - note: always 16-bit */
  zbNwkAddr_t     aSrcAddr;        /* source address or group (never 64-bit) */
  zbEndPoint_t    srcEndPoint;     /* source endpoint */
  zbProfileId_t   aProfileId;      /* profile ID (filtered automatically by APS) */
  zbClusterId_t   aClusterId;      /* cluster ID (no filter applied to clusters) */
  uint8_t         asduLength;      /* length of payload (or this fragment if fragmented) */
  uint8_t         *pAsdu;          /* pointer to payload */
  zbStatus_t      status;          /* note: status field is not presented,
                                      as any packets that fail will not become indications */
  bool_t          fWasBroadcast;  /* was the packet groupcast or broadcast? */
  zbApsSecurityStatus_t fSecurityStatus; /* was the packet secure? At which layers? */
  uint8_t         linkQuality;     /* link quality from network layer */
  zbTime_t        rxTime;          /* vendor specific time packet was received (always 0) */
  zbRxFragmentedHdr_t fragmentHdr;  /* only present if txOptions &
                                      gApsTxOptionFragmentationRequested_c */
} 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
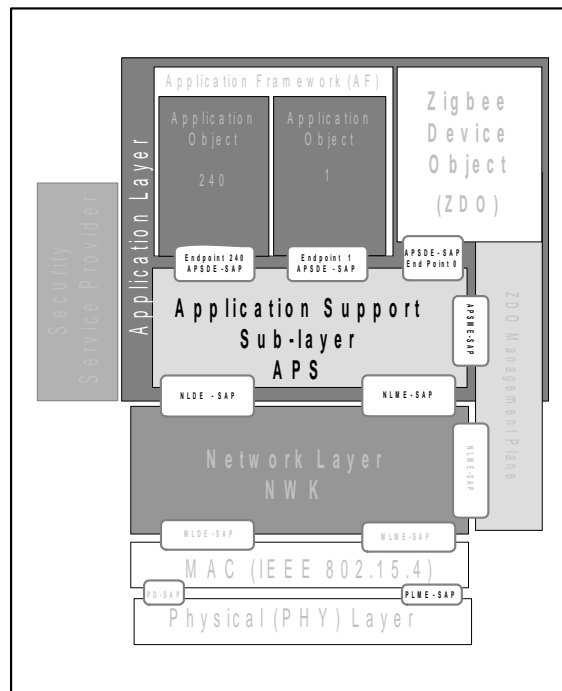
Figure 5-1. BeeStack Application Support Sub-Layer Elements

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():

gApsDesignatedCoordinator_c

ApsUseInsecureJoin_c

gApsChannelMask_c

gApsUseExtendedPANID_c

gApsNonmemberRadius_c

gApsMaxWindowSize_c

BeeStack™ Software Reference Manual for ZigBee 2007, Rev. 1.1

gApsInterframeDelay_c

gApsMaxFragmentLength_c

gApsChannelFailureRate_c

gApsChannelTimer_c

gApsTrustCenterAddress_c   Usually ZC IEEE address

gApsTrustCenterNwkAddress_c   Usually 0x0000 (ZC)

gApsSecurityTimeOutPeriod_c   Timeout for authentication

gApsDeviceKeyPairSet_c

gApsTrustCenterMasterKey_c

gApsTrustCenterLinkKey_c

gApsDefaultKeyType_c

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 Management 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-announce.

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.

BeeStack™ Software Reference Manual for ZigBee 2007, Rev. 1.1

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

BeeStack™ Software Reference Manual for ZigBee 2007, Rev. 1.1

- gZbNotSupported_t if binding is not supported
- gZdoDeviceNotFound_c if device is not found in address map

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
- gZdoDeviceNotFound_c if device is not found in address map

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.
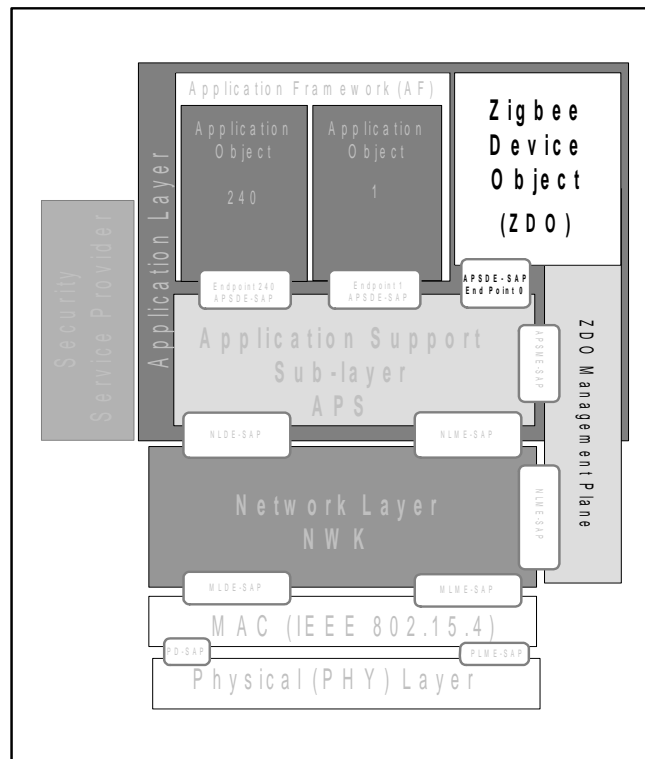
Figure 6-1. ZigBee Device Objects in BeeStack

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 the 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. Figure 6-2 shows the ZDO state machine for BeeStack Codebase version 3.0.0 and higher.
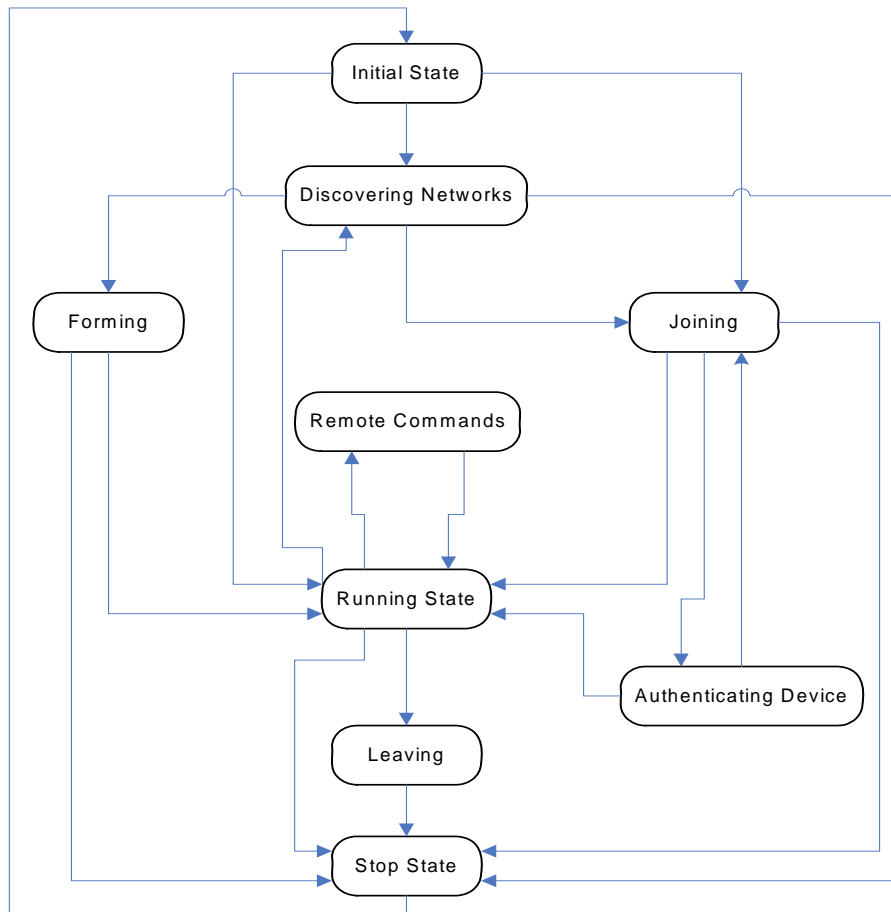
Figure 6-2. ZDO State Machine Overview

State description:

*   The Initial state "gZdoInitialState_c" state is where ZDO always starts (this is off the network)
*   The Discovering Networks state "gZdoDiscoveringNetworksState_c" is where it finds networks. The network layer keeps a copy of the energy scan (ZC only) and list of other ZigBee/802.15.4 devices. The ZDO layer frees this memory when finished. Discovery will continue for a certain amount of time, or when a certain number of tries have been attempted.
*   The Forming state "gZdoFormingState_c" is for ZC and Zx (in coordinator mode) devices. If forming fails, it goes back to discovery state.
*   The Joining state "gZdoJoiningState_c" is for ZR, ZED and Zx (in router/end-device modes) devices. If joining fails it goes back to discovery state.

- The Authenticating Device state "gZdoDeviceAuthenticationState_c" is for security ZRs and ZEDs. If this state fails, goes back to network discovery state.
- The Running state "gZdoRunningState_c" indicates that the device is running and can now send/receive packets.
- The Remote Commands state "gZdoRemoteCommandsState_c" is used for remote devices causing this device to perform things such as a scan for networks, or changing channels. Once complete, this goes back to running state.
- The Leaving state "gZdoLeavingState_c" means this device is cleanly leaving the network, which will allow other devices in the network to clean up their tables automatically. A ZR or ZC may also tell its children to leave.
- The Stop State "gZdoStoppingState_c" means this device is stopping. This cannot fail. The device stops and alternately clears out NVM.

Each state has a number of sub-states caused by "events". See the ZdoStatemachinehandle.h for each state to see what these are.

The overall ZDO process, when ZDO_Start() is called by the application begins with a network discovery [ND] attempt. Once the best selection of beacons are gathered, the form (ZC) or join (ZR, ZED) attempt is made. If that fails, that starts over again, up to a user selected length of time and number of attempts, whichever comes first. See Figure 6-3:

```
+----------------------------- (T) -----------------------------------+
|                                                                     |
[ES][AS](B)[AS](B)[AS][FJ]...(RI)...[ES][AS](B)[AS](B)[AS][FJ]...(RI)....
|                     |
+----------- [ND] -------+
```
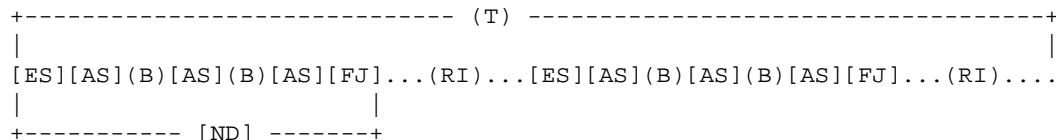
Figure 6-3. ZDO Process

The properties that define the various configurable number of attempts and time-outs, as well as the figure key, are as follows:

(T)	Total Time-out, mDefaultValueOfFormationAttemptsTimeOut_c(ZC), mDefaultValueOfDiscoveryAttemptsTimeOut_c(ZR,ZED) (sec)
[ES]	Energy Scan (ZC only), gScanDuration_c (802.15.4 MAC exponential wait)
[AS]	Active Scan (all device types), gScanDuration_c (802.15.4 MAC exponential wait)
(B)	Time Between Active Scans, mDefaultValueOfNwkTimeBwnScans_c (ms)
[FJ]	Form or join attempt. If this fails, will wait the rejoin interval. If it works, it goes to authenticating/running state.
(RI)	Rejoin interval, mDefaultValueOfRejoinInterval_c (sec), configurable through the SAS and commissioning cluster
[ND]	Network Discovery attempt. This will occur 'N' number of times, where 'N' is mDefaultValueOfNwkFormationAttempts_c or mDefaultValueOfNwkDiscoveryAttempts_c

For more information, see Table 1.48 in the ZigBee Specification: Config_NWK_Scan_Attempts [ZCL Commissioning Cluster], ScanAttempts NlmeGetRequest(gNwkScanAttempts_c).

The mDefaultValueOfNwkScanAttempts_c function describes how many times to discover networks before attempting to form or join. Trying multiple times helps in a noisy or dense network environment.

6.2   General ZDO Interfaces (Codebase Version 3.0.0 and Higher)

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

NOTE
To maintain some backwards compatibility, some older version APIs can still be used.

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 the 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).

gZdoInitialState_c	Initial state
gZdoDiscoveringNetworksState_c	Network discovery in progress
gZdoFormingState_c	Attempting to form a network, only valid for ZC
gZdoJoiningState_c	Attempting to join a network, only valid for ZR, ZED
gZdoOrphanJoiningState_c	Attempting to orphan join a network, only valid for ZR, ZED
gZdoCoordinatorRunningState_c	Network formed, only valid for ZC
gZdoRouterRunningState_c	Network joined, only valid for ZR
gZdoEndDeviceRunningState_c	Network joined, only valid for ZED
gZdoRunningState_c	Device is running, network joined or formed (any device type)
gZdoDeviceAuthenticationState_c	Waiting for authentication to complete only valid for ZR, ZED
gZdoRemoteCommandsState_c	Executing a command received over the air.(ZDP commands)
gZdoDeviceWaitingForKeyState_c	Waiting for a Link Key
gZdoLeavingState_c	Announcing that the device is leaving the network
gZdoStoppingState_c	Stop state, resets stack

6.2.2   Start ZDO State Machine

The function `ZDO_Start()` initializes the device using one of the following start options:

DDDx xxxx   Device type

xxxS Sxxx   Startup set

xxxx xCCC   Startup control mode

Choose one of these, or leave field set to 0 to use previous setting (ignored if not a Combo device):

gZdoStartMode_Zc_c   0xc0 - start as ZC (combo device only)

gZdoStartMode_Zr_c   0x80 - start as ZR (combo device only)

gZdoStartMode_Zed_c   0x20 - start as ZED (combo device only)

gZdoStartMode_ZedRx_c   0x60 - start as RxOnIdle=TRUE ZED (combo device only)

Choose one of these options:

gZdoStartMode_NvmSet_c   0x00 - copy NVM set (if any) to working set, then start. If no NVM, use ROM set.

gZdoStartMode_RomSet_c   0x08 - copy ROM set to working set (factory defaults), then start

gZdoStartMode_RamSet_c   0x10 - use working startup set in RAM.

gZdoStartMode_SasSet_c   0x18 - copy commissioning cluster set to working set, then start. If not valid, use NVM set.

Choose one of these (see also NLME-JOIN.request and zbNwkJoinMode_t in ZigBee.h) T he order of these must match NLME-JOIN.request:

gZdoStartMode_Associate_c   0x00 (default) use association (ZR, ZED only), or form (ZC)

gZdoStartMode_OrphanRejoin_c   0x01 FS specific: use orphan rejoin (ZR, ZED only)

gZdoStartMode_NwkRejoin_c   0x02 use NWK rejoin (ZR, ZED only)

gZdoStartMode_FindAndRejoin_c   0x03 for ZR, ZED only, search for network on this and other channels, then silent join

gZdoStartMode_SilentStart_c   0x04 - already part of the network (no form/join needed)

gZdoStartMode_StartMask_c   0x07 - mask for above fields

Default Mode:

gZdoStartMode_Default_c   (gZdoStartMode_Associate_c | gZdoStartMode_NvmSet_c)

Function

```
void ZDO_Start(ZdoStartMode_t startMode)
```

6.2.3   Stop With Mode Select

This function allows full application control of how to leave the network and/or stop its ZDO state machine.

The following stop modes are available:

stop mode bit mask for ZDO_StopEx():

xxxx xxxA	Announce (leave) before stopping
xxxx xxCx	Remove children if announce leave is active.
Nxxx xxxx	Reset NVM
xRxx xxxx	Restart after stopping
xxBx xxxx	Do not reset binding and group tables

Function

```
void ZDO_StopEx(ZdoStopMode_t stopMode);
```

6.2.4   Stop ZDO State Machine

This function instructs a device to stop its ZDO State machine. It calls ZDO_StopEx(gZdoStopMode_Stop_c). This function is available to maintain backward compatibility.

Macro

```
void ZDO_Stop(void)
```

6.2.5   Stop ZDO and Leave

This function sends the command to leave the network to a device and then stop its ZDO state machine. Calls ZDO_StopEx(gZdoStopMode_Announce_c | gZdoStopMode_ResetTables_c | gZdoStopMode_ResetNvm_c);. This function is available to maintain backward compatibility.

Function

```
void ZDO_Leave(stop)
```

6.3 General ZDO Interfaces (CodeBase Versions Before 3.0.0)

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

6.3.1 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):

gZdoInitialState_c	Initial state
gZdoDiscoveringNetworksState_c	Network discovery in progress
gZdoFormingState_c	Attempting to form a network, only valid for ZC
gZdoJoiningState_c	Attempting to join a network, only valid for ZR, ZED
gZdoOrphanJoiningState_c	Attempting to orphan join a network, only valid for ZR, ZED
gZdoCoordinatorRunningState_c	Network formed, only valid for ZC
gZdoRouterRunningState_c	Network joined, only valid for ZR
gZdoEndDeviceRunningState_c	Network joined, only valid for ZED
gZdoRunningState_c	Device is running, network joined or formed (any device type)
gZdoDeviceAuthenticationState_c	Waiting for authentication to complete only valid for ZR, ZED
gZdoRemoteCommandsState_c	Executing a command received over the air.(ZDP commands)
gZdoDeviceWaitingForKeyState_c	Waiting for a Link Key
gZdoLeavingState_c	Announcing that the device is leaving the network
gZdoStoppingState_c	Stop state, resets stack

6.3.2 Start ZDO State Machine without NVM

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

Macro

#define ZDO_Start(gStartWithOutNvm_c)

6.3.3   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.

gStartSilentRejoinWithOutNvm_c   Allows a device to rejoin a network, does not restore information from its memory, and does not notify other devices of its return to the network.

gStartSilentNwkRejoin_c   Allows a device to rejoin a network, restores information from its memory, and does not notify other devices of its return to the network.

6.3.4   Stop ZDO State Machine

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

Macro

```
#define ZDO_Stop()
```

6.3.5   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()
```

6.4 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.

6.4.1 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.

6.4.1.1 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:

- gStartWithOutNvm_c
- gStartAssociationRejoinWithNvm_c
- gStartOrphanRejoinWithNvm_c
- gStartNwkRejoinWithNvm_c
- gStartSilentRejoinWithNvm_c

6.4.1.2 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

6.4.1.3 ZC Running State

In running machine state, the ZC machine state supports:

- gStop_c
- gKeyTransferInitiated_c
- gManagementCommandSent_c
- gChildLeaveSuccess_c

6.4.1.4 ZC Key Transfer state

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

- gKeyTransferSuccess_c

6.4.1.5 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

6.4.1.6 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

6.4.2 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.

6.4.2.1 ZR Initial Machine State

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

- gStartWithOutNvm_c
- gStartAssociationRejoinWithNvm_c
- gStartOrphanRejoinWithNvm_c
- gStartNwkRejoinWithNvm_c
- gStartSilentRejoinWithNvm_c

- gStartSilentRejoinWithOutNvm_c
- gStartSilentNwkRejoin_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.

6.4.2.2   ZR 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

6.4.2.3   ZR 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

6.4.2.4   ZR Running State

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

ZR running state supports these events:

- gStartDevice_c
- gKeyTransferInitiated_c
- gStartSilentNwkRejoin_c
- gStop_c
- gAnnceStop_c
- gChildLeaveSuccess_c
- gManagementCommandSent_c

6.4.2.5   ZR 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

BeeStack™ Software Reference Manual for ZigBee 2007, Rev. 1.1

6.4.2.6   ZR Stop State Machine

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

ZR stop state machine supports the following event:

- gStop_c

6.4.3   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.

6.4.3.1   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

6.4.3.2   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

6.4.3.3   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

6.4.3.4   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

6.4.3.5   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

6.4.3.6   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

6.4.3.7   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

6.4.3.8   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

6.4.3.9   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

6.5   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

BeeStack™ Software Reference Manual for ZigBee 2007, Rev. 1.1

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);
```

BeeStack™ Software Reference Manual for ZigBee 2007, Rev. 1.1

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 Network Update Request

A local device requests that a remote device update its network information or request the remote node to perform and energy scan or move to a different channel using the Mgmt_nwk_update_req command. Generated by a management application, the Mgmt_nwk_req sends the request to a remote device. The remote device executes the request using the parameters supplied in the Mgmt_nwk_update_req. The local device is notified of the results when a Mgmt_nwk_update_Notify is received.

Prototype

```
void ASL_Mgmt_NWK_Update_req(aDestAddress,aChannelList,iScanDuration,iScanCount)
```

ZDP Returns

Nothing

7.8.8   Management Network Update Notify

If a local device has received an Mgmt_nwk_update_req, the local device can notify the remote device of of the results by sending a Mgmt_nwk_update_Notify

Prototype

```
voidASL_Mgmt_NWK_Update_Notify(aDestAddress, aScannedChannels, iTotalTransmissions, \
                     iTransmissionFailures, iScannedChannelListCount,paEnergyVslues, status)
```

ZDP Returns

Nothing

7.8.9   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
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
gZdoNoDescriptor_c	0x89	Child descriptor was not available following a discovery request to a parent
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
gZdoNotAuthorized_c	0x8d	Permission configuration table on the target indicates that the request is not authorized from this device

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.
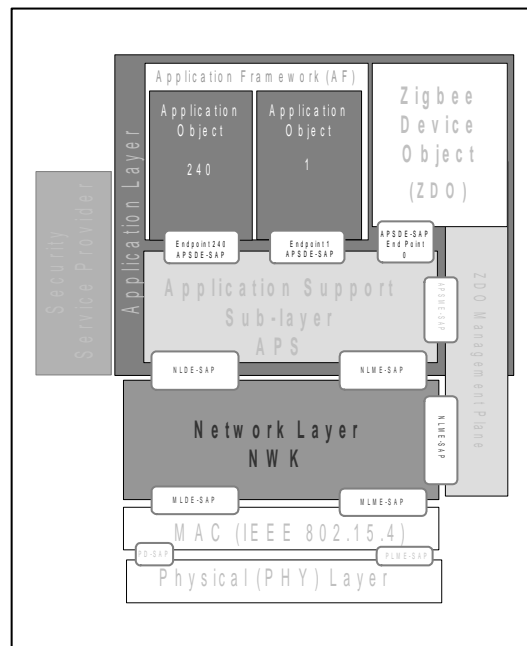
Figure 8-1. Network Layer Interfaces

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.

3 2 2 2 1 1 0 0 0 1 8 4 0 6 2 8 4 0		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
nwkPANId	0x80	16-bit PAN ID	0x0000 - 0xffff	macPANId	0xffff
nwkSequenceNumber	0x81	Integer	0x00 - 0xff	Sequence number used to identify outgoing frames	Random value from within range
nwkPassiveAckTimeout	0x82	Integer	0x0000 - 0x2710	Maximum time duration in milliseconds allowed for parent and all child devices to retransmit a broadcast message (passive ACK time-out)	Defined in the stack profile
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	Defined in the stack profile
nwkMaxDepth	0x85	Integer	0x00 - 0xff	Depth a device can have: maximum hops from ZC	Defined in stack profile
nwkMaxRouter	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	Defined in stack profile
nwkNeighborTable	0x87	Set	Variable	Current set of neighbor table entries in the device	Null set
nwkNetworkBroadcastDeliveryTimel	0x88	Integer	(nwkPassiveAckTimeouT* nwkBroadcastRetries 0x00 – 0xff	Time duration in seconds that a broadcast message needs to encompass the entire network	nwkPassiveAckTimeout * nwkBroadcastRetries
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

Table 8-3. NWK Information Base Attributes (continued)

Attribute	ID	Type	Range	Description	Default
nwkTimeStamp	0x8c	Boolean	TRUE or FALSE	Flag to determine whether a time stamp indication is provided on incoming and outgoing packets. TRUE: time indication provided FALSE: no time indication provided	FALSE
nwkTxTotal	0x8d	Integer	0x0000 - 0xffff	Count of unicast transmissions made by the NWK layer on the device	0x00
nwkSymLink	0x8e	Boolean	TRUE or FALSE	TRUE: routes are considered to be comprised of symmetric links. FALSE: routes are not considered to be comprised of symmetric links.	FALSE
nwkCapabilityInformation	0x8f	Bit vector	N/A	Device capability information established at network joining time	0x00
nwkAddrAlloc	0x90	Integer	0x00 - 0x02	0x00 = Use distributed address allocation 0x01 = reserved 0x02 = use stochastic address allocation	0x00
nwkUseTreeRouting	0x91	Boolean	TRUE or FALSE	TRUE: assume the ability to use hierarchical routing. FALSE: never use hierarchical routing.	TRUE
nwkManagerAddr	0x92	Integer	0x0000 - 0xfff7	The address f the designated network channel manager function. Usually the ZC.	0x0000
nwkMaxSourceRoute	0x93	Integer	0x00 - 0xff	The maximum number of hops in a source route	0x0c
nwkUpdateId	0x94	Integer	0x00 - 0xff	Value identifying a snapshot of the network settings with which the node is operating with	0x00
nwkTransactionPersistenceTime	0x95	Integer	0x0000 - 0xffff	Maximum time (in superframe periods) that a transaction is stored by a ZC and indicated in its beacon	0x01f4

BeeStack™ Software Reference Manual for ZigBee 2007, Rev. 1.1

Table 8-3. NWK Information Base Attributes (continued)

Attribute	ID	Type	Range	Description	Default
nwkShortAddress	0x96	Integer	0x0000 - 0xfff7	16-bit address that the devices uses to communicate with the PAN	0xffff
nwkStackProfile	0x97	Integer	0x00 - 0x0f	Identifier of the ZigBee stack profile in use for the device	
nwkBroadcastTransactionTable	0x98	Set	N/A	Current set of broadcast transaction table entered in the device	Null set
nwkGroupIDTable	0x99	Set	Variable	Set of group identifiers for groups of which this device is a member	Null set
nwkExtendedPANID	0x9a	64-bit extended address	0x0000000000 000000 - 0xfffffffffffffffe	Extended PAN Identifier for the PAN of which the device is a member	0x0000000000 000000
nwkUseMulticast	0x9b	Boolean	TRUE or FALSE	TRUE: multicast occurs at the network layer FALSE: multicast occurs at the APS layer and using the APS header	TRUE
nwkRouteRecordTable	0x9c	Set	Variable	Route record table	Null set
nwkIsConcentrator	0x9d	Boolean	TRUE or FALSE	TRUE: Device is a concentrator FALSE: Device is not a concentrator	FALSE
nwkConcentratorRadius	0x9e	Integer	0x00 - 0xff	Hop count radius for concentrator route discoveries	0x00
nwkConcentratorDiscoveryTime	0x9f	Integer	0x00 - 0xff	Time in seconds between concentrator route discoveries. If set to 0x00, the discoveries are done at startup and by the next higher layer only.	0x00
nwkSecurityLevel	0xa0	octet	0x00-0x07	The security level for outgoing and incoming NWK frames; the allowable security level identifiers	set by stack profile - normally 0x05

Table 8-3. NWK Information Base Attributes (continued)

Attribute	ID	Type	Range	Description	Default
nwkSecurityMaterialSet	0xa1	security descriptor	variable	Set of network security material descriptors capable of maintaining an active and alternate network key.	NA
nwkActiveKeySeqNumber	0xa2	integer	octet	The sequence number of the active network key in *nwkSecurityMaterialSet.*	0x00
nwkAllFresh	0xa3	Boolean	TRUE or FALSE	Indicates whether incoming NWK frames must be all checked for freshness when the memory for incoming frame counts is exceeded.	TRUE
nwkSecureAllFrames	0xa5	Boolean	TRUE or FALSE	Indicates whether security shall be applied to incoming and outgoing NWK data frames. If set to 0x01, security processing shall be applied to all incoming and outgoing frames except data frames destined for the current device that have the security sub-field of the frame control field set to 0. If this attribute has a value of 0x01, the NWK layer shall not relay frames that have the security sub-field of the frame control field set to 0. The SecurityEnable parameter of the NLDEDATA.requestprimitive shall override the setting	TRUE
nwkLinkStatusPeriod	0xa6	Integer	0x00 - 0xff	The link time in seconds between link status command frames	0x0f
nwkRouterAgeLimit	0xa7	Integer	0x00 - 0xff	Number of missed link status command frames before resetting the link costs to zero	0x03
nwkUniqueAddr	0xa8	Boolean	TRUE or FALSE	Flag that determines whether the NWK layer should detect and correct conflicting addresses. TRUE: assume addresses are unique FALSE: addresses may not be unique	TRUE

Table 8-3. NWK Information Base Attributes (continued)

Attribute	ID	Type	Range	Description	Default
nwkAddressMap	0xa9	Set	Variable	Current set of 64-bit IEEE to 16-bit network address map	Null set
nwkProtocolVersion	N/A	octet	0x02	The version of the ZigBee NWK protocol in the device	0x02
nwkTxTotalFailures	N/A	octet	00-0xFF	Number of Tx failures totally registered	NA
nwkIeeeAddress	NA	IEEE address	IEEE address	The device own IEEE address	NA
nwkLogicalChannel	NA	octet	11-26	Channel currently used	NA
nwkParentShortAddress	NA	integer	0x0000-0xFFFF	Parent's short address	NA
nwkParentLongAddress	NA	IEEE address	IEEE address	Parent's IEEE address	NA
nwkDeviceDepth	NA	octet	0x0-0x0f	Current depth of the device	NA
NwkKeyType	NA	Octet	0x01, 0x05	NetworkKeyType	0x01
nwkPreconfiguredKey	NA	bool_t	TRUE FALSE	Is Pre-configured key used?	NA
nwkDevType	NA	octet	0x00 0x01 0x02	Coordinator = 0x00 Router = 0x01 End device 0x02	NA

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;
```

BeeStack™ Software Reference Manual for ZigBee 2007, Rev. 1.1

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);
```

BeeStack™ Software Reference Manual for ZigBee 2007, Rev. 1.1

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 16-bit data from over-the-air to native format
uint16_t Native2OTA16(uint16_t);	For converting 16-bit data from native to over-the-air format
uint32_t OTA2Native32(uint32_t);	For converting 32-bit data from over-the-air to native format
uint32_t Native2OTA32(uint32_t);	For converting 32-bit data from native to over-the-air format
uint64_t OTA2Native64(*uint64_t);	For converting 64-bit data from over-the-air to native format
uint64_t Native2OTA64(*uint64_t);	For converting 64-bit data from native to over-the-air format

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

Function	Description
bool_t IsSelfIeeeAddress(zbIeeeAddr_t aleeeAddr);	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 before sending
void BeeUtilLargeMemSet(void *pBuffer, uint8_t value, uint16_t iCount);	Set a large array of memory with the given value (larger than FLib_memset can handle)
void BeeUtilLargeZeroMemory(void *pBuffer, uint16_t iCount)	Set a large array of memory to zeroes (larger than FLib_memset can handle)
void Copy8Bytes(zbIeeeAddr_t aleeeAddr1,zbIeeeAddr_t aleeeAddr2	Copies 8 bytes (e.g. IEEE address) from one location to another. Assumes they do not overlap.
bool_t IsEqual8Bytes(zbIeeeAddr_t aleeeAddr1, zbIeeeAddr_t aleeeAddr2);	Are the two IEEE addresses equal?
void Fill8BytesToZero(zbIeeeAddr_t aleeeAddr1);	Fill IEEE (long) address with 0s
void FillWithZero(void *pBuffer, uint8_t size);	Fill any length buffer with 0s
bool_t Cmp8BytesToZero(zbIeeeAddr_t aleeeAddr1);	Is this IEEE address all 0s?
bool_t Cmp8BytesToFs(zbIeeeAddr_t aleeeAddr1);	Is this IEEE address 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
bool_t IsEqual2BytesInt(void *ptr, uint16_t val)	Is the given memory location equal to the value given?
void Set2Bytes(void *ptr, uint16_t val)	Copies a 16-bit variable to a given location with the bytes swapped
void Copy16Bytes(void *pDst, void *pSrc)	Copies 16-bytes of data unconditionally
bool_t Cmp16BytesToZero(void *pArray)	Are all the bytes equal to zero in the given 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
uint8_t BeeUtilClearIndexedBit(uint8_t *pBitArray, index_t iBit);	Clear an indexed bit
bool_t BeeUtilGetIndexedBit(uint8_t *pBitArray, index_t iBit);	Get and indexed bit

Table 10-3. BeeStack Common Network Functions (continued)

Function	Description
bool_t BeeUtilArrayIsFilledWith(uint8_t *pArray, uint8_t value, index_t iLen);	Check to see if an array is filled with a particular value
bool_t ApsIsGroupInGroupTable(zbGroupId_t aGroupId)	Is the given GroupID in the group table?
bool_t IsIncompleteBindNwkAddr(zbNwkAddr_t aNwkAddr)	Is this an incomplete binding network address?
bool_t IsValidPanId(zbPanId_t aPanId)	Is this a valid PAN ID?
bool_t IsValidExtendedPanId(zbIeeeAddr_t aIeeeAddr)	Is this a valid Extended PAN ID?
void BeeUtilZeroMemory(void *pPtr, zbSize_t bufferSize);	Clears a buffer to zero
uint32_t Swap4Bytes(uint32_t iOldValue)	Swaps bytes to convert between OTA and native format for a 32-bit word
void Swap8Bytes(uint8_t *pInput)	Swaps bytes to convert between OTA and native format for a 64-bit word
uint8_t BeeUtilBitToIndex(uint8_t *pBitArray, index_t iLen)	Determine the first bit in a bit indexed array
uint8_t GetRandomRange(uint8_t low, uint8_t high)	Get a number at random from a given range
uint32_t GetRandomNumber(void)	Get a 32-bit random number (available only for the MC1322x)
uint16_t GetRandomNumber(void)	Get a 16-bit random number (not available for the MC1322x)
void SeedRandomNumber(uint16_t seed);	Set a 16-bit seed value
bool_t CanIHearThisShortAddress(zbNwkAddr_t aSourceAddress)	Is the given address within the ICanHearYouTable? TRUE: address is within the table (also returns TRUE if global variable gICanHearYouCounter is equal to zero) FALSE: address is not within the table and gICanHearYouCounter is different than 0

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 ZigBee End Device to poll for joining a network.
gRxOnWhenIdle_d	Set to 0x01 (TRUE) to enable a ZED to continuously power it's receiver. When set to TRUE, the ZED will not 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.

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

Option	Description
mDefaultValueOfEndDeviceBindTimeOut_c	The time (in milliseconds) for an end-device-bind to timeout on the ZC.
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 0x1388 (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. In the case of routers, hoe many failures before attempting to find a new route.
mDefaultValueOfNwkFormationAttempts_c	How many times to scan until a suitable free channel/PAN ID is found. Valid for ZCs only.
mDefaultValueOfDiscoveryAttemptsTimeOut_c	Timeout between network discovery attempts. Valid for ZRs and ZED
mDefaultValueOfNwkDiscoveryAttempts_c	Number of attempts to discover a network to join. Valid for ZRs and ZEDs. Defaults to 0, which means forever.
mDefaultValueOfBatteryLifeExtension_c	Does this node operate on batteries?
mDefaultValueOfCurrPowerSourceAndLevel_c	What is 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	Security level, as defined by the ZigBee specification. To be compliant, security level 5 must be selected. All others do not conform to ZigBee profiles.
mDefaultValueOfLpmStatus_c	Enable low power during startup or wait until application enables low power.
gPowerSource_d	Indicates the power source.
gMinNumberOfRouters_c	The amount of routers to select by default from the NT to be used during the FA scanning procedure
gMaxIncomingErrorReports_c	The minimum amount of reports to receive before start the FA procedure
gMaxTimeoutForIncomingErrorReports_c	The number of minutes to wait between reports to the network manager that this node has reached the maximum # of errors.
gMaxTxFailuresPercentage_c	The limit on the transmissions failure tolerance. Expressed in percentage
gApsInterframeDelayDefault_c	Delay (in milliseconds) between frames when transmitting fragmented packets
gApsWindowSizeDefault_c	This is the # of fragments in a single window

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

Option	Description
gApsMaxFragmentLengthDefault_c	Maximum size of each fragment over the air in fragmented APSDE-DATA.request transmissions
gApsFragmentationPollTimeOut_c	This value is used for increasing the poll rate during the reception of fragmented packets
gApsChannelTimerDefault_c	Timer that counts down to 0 to indicate whether it is OK to change channels. Units in hours
gDefaultScanDuration_c	Sets the default scan duration time for frequency agility
gDefaultScanCount_c	Sets the default scan count for frequency agility
gExtraTimeWindow_c	Sets the default extra time window for frequency agility beyond the scan time per channel. In milliseconds
gFullChannelList_c	Defines which channels to scan when forming or joining a network if the preferred (mDefaultValueOfChannel_c) didn't work.
mDefaultValueOfConfirmationPollTimeOut_c	After a ZigBee End-Device (ZED) receives data from it's parent, how quickly does it poll the parent again for data?
mDefaultValueOfIndirectPollRate_c	How quickly does a ZED poll its parent under normal conditions.
mDefaultValueOfNwkKeyType_c	The default type of key to handle in the secure environment for the NWK layer auxiliary frame.
mDefaultValueOfNwkActiveKeySeqNumber_c	Default value of Network Active Key Sequence Number. Always starts at 0.
mDefaultValueOfTrustCenterKey_c	Default trust center key.
mDefaultValueOfTrustCenterKeyType_c	The default type of key to handle in the secure environment for the APS layer auxiliary frame.
mDefaultValueOfApplicationKeyType_c	The default key type to answer to a Request key.
mDefaultValueOfTrustCenterLongAddress_c	Default value of the Trust Center IEEE address.
mDefaultApsSecurityTimeOutPeriod_c	Default period of time a device will wait for an expected security protocol frame (in milliseconds).
gIeeeFilterMask_c	Allows a router or coordinator to refuse other nodes to join if the IEEE address does not match the filter criteria.
gIeeeFilterValue_c	Allows a router or coordinator to refuse other nodes to join if the IEEE address does not match the filter criteria.
mDefaultValueOfNwkUseMulticast_c	Allows an application to decide if ZigBee Pro networks use multicast (NWK layer groups) or groupcast (APS layer groups).
mDefaultValueOfApsFlagsAndFreqBand_c	Set to 0x40 to indicate 2.4GHz band. Do not change.
mDefaultValueOfManfCodeFlags_c	16-bit number obtained by each OEM in the ZigBee Alliance to uniquely identify that manufacturer.
mDefaultValueOfMaxTransferSize_c	Indicates how large of a ASDU (application payload, including fragmentation) can be received by this node.
mDefaultValueOfParentLinkRetryThreshold_c	How many times a ZED should attempt to contact his parent before initiating rejoin process?

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

Option	Description
mDefaultValueOfCurrModeAndAvailSources_c	Indicates if this devices is mains powered or not in Node Descriptor.
mDefaultValueOfRejoinInterval_c	How often in seconds a ZED will try to rejoin the network after it has left it. This is also used for ZCs and ZRs to determine how long to wait between form/join attempts.
mDefaultValueOfMaxRejoinInterval_c	Upper bound on Rejoin Interval (OrphanScanAttemptsTimeOut), in seconds.
mDefaultValueOfFormationAttemptsTimeOut_c	How long to keep trying to join a network? That is, how many total seconds before giving up? Will give up based on time or time of # attempts, whichever comes first.
mDefaultValueOfOrphanScanAttemptsTimeOut_c	How long to keep trying to rejoin a network? That is, how many total seconds before giving up? Will give up based on time or time of # attempts, whichever comes first.
mDefaultValueOfNwkAllFresh_c	nwkAllFresh field in NIB. Set to TRUE to check all frames for freshness.
mDefaultValueOfNwkSecureAllFrames_c	nwkSecureAllFrames in the NIB. If so, then frames are decrypted and re-encrypted per hop (which allows authentication). If not, then frames are only secured at either end. Must be TRUE to be ZigBee compliant.
mNwkRptConstantCost_c	If this is set to 0x00, the NWK layer shall calculate link cost from all neighbor nodes using the LQI values reported by the MAC layer. Otherwise it shall report a constant value.
mNwkRouteDiscRetriesPermitted_c	The number of retries allowed after an unsuccessful route request.
mNwkUseTreeRouting_c	A flag that determines whether the NWK layer should assume the ability to use tree routing. Stack profile 0x01 (aka ZigBee 2007) uses Tree. Stack profile 0x02 (aka ZigBee Pro) does not.
mDefaultNwkNextAddress_c	Only relevant if mNwkAddressAlloc_c is FALSE. The next network address that will be assigned to a device requesting association.
mDefaultNwkAvailableAddress_c	Only relevant if mNwkAddressAlloc_c is FALSE. The size of remaining block of addresses to be assigned.
mDefaultNwkAddressIncrement_c	Only relevant if mNwkAddressAlloc_c is FALSE. The amount by which gDefaultNwkNextAddress_c increments each time an address is assigned.
mDefaultNwkShortAddress_c	16-bit address that the device uses to communicate with the PAN.
mNwkProtocolVersion_c	The version of the ZigBee protocol currently in use by the NWK layer. Must be 0x02. Do not change.
mDefaultLogicalChannel_c	Default value for the logical channel. Set to 0x00 to allow the node to pick a channel from the channel list.
mDefaultParentShortAddress_c	Default value of current device's parent address, little endian. Set to 0xffff to find parent.
mDefaultParentLongAddress_c	Default value of current device's parent long address, little endian.
mDefaultValueOfComplexDescFieldCount_c	Default value for field count. Indicates how many fields are included in the complex descriptor

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

Option	Description
mDefaultValueOfComplexDescLangCharSet_c	Default values for Language code and character set in Complex descriptor.
mDefaultValueOfComplexDescManufactureName_c	Default value of manufacturer name in complex descriptor.
mDefaultValueOfComplexDescModelName_c	Default value of model name in complex descriptor.
mDefaultValueOfComplexDescSerialNumber_c	Default value of serial number in complex descriptor.
mDefaultValueOfComplexDescDeviceUrl_c	Default value of device URL in complex descriptor.
mDefaultValueOfComplexDescIcon_c	Default value of descriptor Icon in complex descriptor.
mDefaultValueOfComplexDescIconUrl_c	Default value of Descriptor Icon URL in complex descriptor.
gDefaultRetriesStochasticAddrLocalDevice_c	Default value of the times in which a ZC or ZR can generate a network address for the device who attempt to join to it.
gDefaultRetriesStochasticAddrNetworkWide_c	Number of times a parent will try to assign a random address to a joining child.
gApsDesignatedCoordinatorDefault_c	Set to TRUE (1) if the device should become the ZigBee Coordinator on startup, FALSE if otherwise. This flag is only relevant for a ZigBee device that can become a router or coordinator on startup.
gApsNonMemberRadiusDefault_c	Value to be used for the NonmemberRadius parameter when using NWK layer multicast.
gApsUseInsecureJoinDefault_c	Flag controlling insecure join at startup. Specifies if the rejoin process should be secured or unsecured.
gNwkManagerShortAddr_c	Short address of the network manager.
gConcentratorRadius_c	The radius for many-to-one route discovery from a concentrator (gateway).
gConcentratorDiscoveryTime_c	The amount of time (in seconds) between many-to-one route discoveries.
gApsMaxEntriesForPermissionsTable_c	Maximum number of entries for the Permissions Configuration Table, a zero value means that the Permissions Configuration Table is not available.
gZdoStopModeDefault_c	Stop mode to use if this node receives an over-the-air NLME-LEAVE command.
gZdoStartModeDefault_c	Start mode to use if this node receives an over-the-air NLME-LEAVE command.

11.2 More Compile-time Options

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

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

Option	Description
gNwkInfobaseMaxBroadcastRetries_c	Number of retries on each broadcast. Default is 2.
gCoordinatorNwkInfobaseMaxNeighborTableEntry_c	Maximum number of neighbor table entries on a ZC. Default is 24.
gRouterNwkInfobaseMaxNeighborTableEntry_c	Maximum number of neighbor table entries on a ZR. Default is 25.
gEndDevNwkInfobaseMaxNeighborTableEntry_c	Maximum 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 15 in stack profile 0x02. 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	Limits which nodes the NWK layer can hear. Use to analyze multi-hop behavior of the network.
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).
gFrequencyAgilityCapability_d	Enables the Frequency Agility state machine.
gFragmentationCapability_d	Enables the Fragmentation feature.
gInterPanCommunicationEnabled_c	Enables Inter-Pan communication features.
gHttMaxIndirectEntries_c	Max number of indirect messages that can be in the Handle Tracking Table
gMaxNumberOfTxAttempts_c	How many messages will be sent to the MAC for polling ZigBee End-Devices
gSecNwkIncFrameCounters_c	Incoming frame counters. This determines the number of secure nodes this node may communicate with

Table 11-2. BeeStackConfiguration.h Compile-Time Options (continued)

Option	Description
gApsMaxDataHandlingCapacity_c	Determines the # of simultaneous messages APS can handle from higher layers (both ZDP and application).
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.
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.
gDevice_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.

Table 11-2. BeeStackConfiguration.h Compile-Time Options (continued)

Option	Description
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.
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.

Table 11-2. BeeStackConfiguration.h Compile-Time Options (continued)

Option	Description
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.
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.
gMgmt_NWK_Update_req_d	Enable request
gMgmt_NWK_Update_notify_d	Enable response
gMSPstackProfileEnabled_d	Enables manufacturer specific profile ID.
gZigbeeProIncluded_d	Enables Stack Profile 2 for the application.
gAppStackProfile_c	Stack profile value over the air.
gMaxBroadcastTransactionTableEntries_c	Number Of Outstanding Broadcasts that are supported.
gNwkInfobasePassiveAckTimeout_c	Network Passive Ack Timeout.
gNwkInfobaseBroadcastDeliveryTime_c	Time duration in seconds that a broadcast message needs to encompass the entire network.
gNwkAgingTimeForEndDevice_c	Time duration in minutes that a neighbor table entry for an end device needs to wait to be expired.
gNwkAgingTimeForRouter_c	Time duration in minutes that a neighbor table entry for a router needs to wait to be expired.
gNwkTransPersistenceTime_c	This value is used to purge the data after specified duration. Do not change.
gNwkRoutingMaxRouteDiscoveyTableEntry_c	How many simultaneous route discoveries can this node support?
gNwkInfobaseMaxSourceRouteTableEntry_c	How many route record entries can this node sustain?
gNwkMaxHopsInSourceRoute_c	The maximum number of hops in a source route.

BeeStack™ Software Reference Manual for ZigBee 2007, Rev. 1.1

Table 11-2. BeeStackConfiguration.h Compile-Time Options (continued)

Option	Description
gConcentratorFlag_d	Can this node be a concentrator? Only available on Stack Profile 0x02.
gNwkHighRamConcentrator_d	Allows the storing/retrieving source routes.
gHandleTrackingTableSize_c	Max Handle Tracking Table Entries.
gPacketsOnHoldTableSize_c	How many simultaneous messages the network layer can hold until they get routed.
gMaxNwkLinkRetryThreshold_c	Default Transmit Failure Counter. Indicates how many times to fail a transmission to a specific device before a route repair is initiated
gSamePanIdOk_c	It is allowed in the ZigBee specification to form a network with the same PAN ID as an existing network; as long as the extended PAN ID is different. Disallow this by default.
gDefaultValueOfMaxEntriesForExclusionTable_c	Default number of entries in exclusion table.
gApsLinkKeySecurity_d	APS Security (Optional Stack profile 0x01 and 0x02).
gSKKESupported_d	APS SKKE activated or not?
gBindCapability_d	Is binding capability enabled?
gApsAckCapability_d	Does APS Support ACK (acknowledgements)?
gNetworkManagerCapability_d	This flag determines if the Node is a NwkManager or not.
gConflictResolutionEnabled_d	Is Conflict Resolution Capability enabled?
gNetworkLinkStatusPeriod_c	The time in seconds between link status command frames.
gNetworkRouterAgeLimit_c	The number of missed link status command frames before resetting the link costs to zero.
gNwkLinkStatusMultipleFramesPeriod_c	Time (in ms) between split link status frames if they need to span multiple packets due to a large number of neighbors.
gApsMaxAddrMapEntries_c	Maximum number of Address Map entries.
gApsMaxLinkKeys_c	Maximum number of link keys. This value should match gApsMaxAddrMapEntries_c.
gApscMinDuplicationRejectionTableSize_c	Keeps track of previously heard APS frames so they are only sent to the application once (prevents duplicates).
gAps64BitAddressResolutionTimeout_c	Wait (in ms) duration for the expected delivery time of the nwk address response needed to complete the address map.
gDefaulEntriesInSKKEStateMachine_c	Specifies the number of simultaneous SKKE processes that can be handled at the same time with different devices.
gDefaultEntriesInEAStateMachine_c	Specifies the number of simultaneous Entity Authentications that can be handled at the same time with different devices.
gExtended_Simple_Desc_req_d	Enable request.
gExtended_Simple_Desc_rsp_d	Enable response.
gExtended_Active_EP_req_d	Enable request.

Table 11-2. BeeStackConfiguration.h Compile-Time Options (continued)

Option	Description
gExtended_Active_EP_rsp_d	Enable response.
gUser_Desc_req_d	Enable request.
gUser_Desc_rsp_d	Enable response.
gBkup_Discovery_cache_d	Enable Backup Discovery Cache Capability?
gNwkEnergyLevelThresHold_d	Energy Threshold used when performing ED Scan. 0xFF means all noise levels are accepted.

Chapter 12
BeeStack Security

BeeStack supports full ZigBee security for Stack Profile 0x01 and Stack Profile 0x02 (MC1322x only) of the ZigBee 2007 specification.

12.1 Security Overview

Adding security into a ZigBee network has the following effects

- Network (NWK) layer security for network command frames (route request, route reply, route error) (ZigBee Feature Set, ZigBee Pro feature set)
- Application (APL) layer security for Application Support Sub-layer (APS) frames (Optional for ZigBee and ZigBee Pro feature set.)
- Entity authentication.
- 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.
- Transmit speed becomes slightly slower (by about 5ms per encode/decode).

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

- 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.)
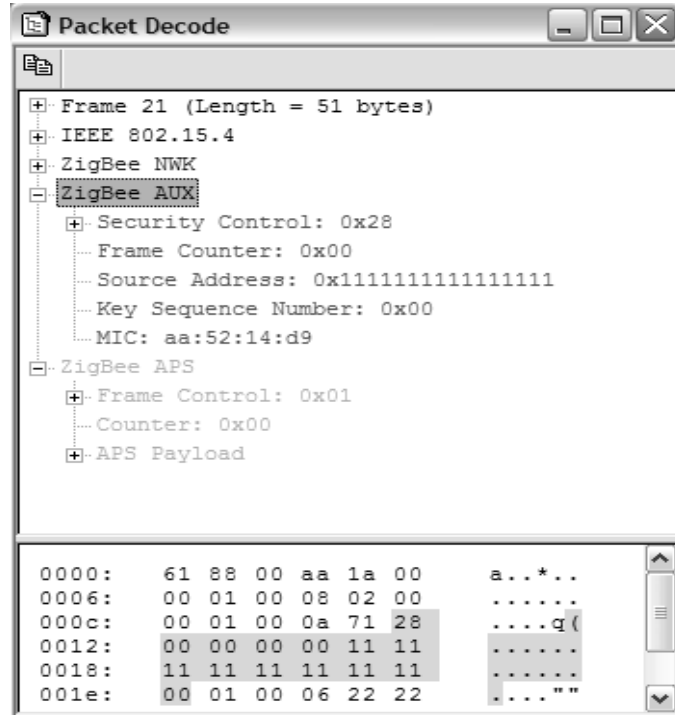
Figure 12-1. Secure ZigBee Packet Example

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.

12.1.1 Security Modes

Two security modes are available:

- Standard Mode (ZigBee and ZigBee Pro feature sets)
    - Two NWK keys and APL security via NWK key.
    - Ability to switch NWK keys.
    - Optional use of Application Link Keys for pairs of communicating devices at APL.
- High Security Mode (ZigBee Pro Feature Set only)
    - Two NWK keys
    - Separate Link Keys for pairs of communicating devices at Application layer.
    - Master Keys with the Trust Center for key transport and key establishment.
    - Ability to switch NWK keys.
    - Entity authentication between all pairs of communicating devices.

12.2   Security Implementation

The Trust Center creates and distributes the network and link keys. It also manages the switch from active to secondary network keys (Standard Security mode or High Security mode).

Optional support is included for Master Keys and Trust Center Link Key establishment and transport. (Optional in Standard Security mode and mandatory in High Security mode.)

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.

In addition to standard and high ZigBee security, BeeStack can 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, deselect the gApsTxOptionNoSecurity_c field in the txOptions flags of the AF_DataRequest() and set the property gAllowNonSecure_d to TRUE.

12.3   Security Configuration Properties

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

NOTE
Select the security type (non, standard, high) choosing the appropriate stack library configuration in BeeKit.

12.3.1   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).

12.3.2   mDefaultValueOfNwkSecurityLevel_c

Always use mDefaultValueOfNwkSecurityLevel_c level 5 for compatibility with ZigBee stack profile 0x01. The other ZigBee security levels are covered later in this section.

12.3.3   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).

12.3.4   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.

12.3.5   mDefaultValueOfNwkKeyType_c

The default type of key to handle in the secure environment for the NWK layer auxiliary frame. Refers to mDefaultValueOfNetworkKey_c. Set to 0x05 for high security systems, set to 0x01 for standard security systems.

12.3.6   mDefaultValueOfTrustCenterKeyType_c

Default trust center key. This key can be used as master or link key, the type of key is determined by mDefaultValueOfTrustCenterKeyType_c

12.3.7   mDefaultValueOfTrustCenterKeyType_c

The default type of key to handle in the secure environment for the APS layer auxiliary frame.

- • KeyType = 0x00 Trust Center master key.
- • KeyType = 0x04 Trust Center link key.
- • KeyType = 0xFF No key preconfigured.

12.3.8   mDefaultValueOfTrustCenterKey_c

Default value of the Trust Center IEEE address. Must be preconfigured if Link keys or high security are used. Can be set when the TC is know but it will be changed by the transport key if standard security only is used.

12.3.9   mDefaultValueOfApplicationKeyType_c

The default key type to answer to a Request key when answering 0x02, which means Application Master key or application link key.

- • 0x02 = gApplicationMasterKey_c
- • 0x03 = gApplicationLinkKey_c

12.3.10   gSKKESupported_d

APS SKKE enabled or not, the use of SKKE help to establish link keys between the nodes through over the IAR commands (Optional in Stack Profile 0x01 and 0x02).

12.3.11   gApsLinkKeySecurity_d

- • APS Security (Optional Stack profile 0x01 and 0x02) enabled
- • mDefaultValueOfTrustCenterLongAddress_c

12.4   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 bool_t deviceInExclusionTable(uint8_t *pIeeeAddress) function 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 the `ZdoNwkManager.c` file.

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
);
```

12.5   IEEE address join filter

BeeStack has an IEEE address join filter feature built in that works independently of security mode (non, standard, high). The IEEE filter allows a router or coordinator to refuse to allow other nodes to join if the IEEE address does not match the filter criteria. Set both the gIeeeFilterMask_c and gIeeeFilterValue_c properties. The filter check function is available in source code: See ValidateIeeeAddress() in the `AppStackImpl.c` file.

Chapter 13
Permission Configuration Table

This chapter describes the optional BeeStack Permission Configuration Table feature. The Permission Configuration Table feature allows out of band configuration so that designated devices can access a particular functionality in the stack.

NOTE
The Permission Configuration Table only is applicable in stack configurations that employ security.

See Chapter 4.6.3.8 of revision 17 in the ZigBee Specification.for more details about the Permission Configuration Table.

13.1   Permission Configuration Table API

The Permission Configuration table is found in the `AppStackImpl.c` file.

13.1.1   AddDeviceToPermissionsTable

Adds a device to the Permission Configuration Table, where aDevAddr is the IEEE device address, and permissionCategory is the bit mask representing the device permissions. Adding a device in this table also adds the device to the address map.

```
uint8_t  AddDeviceToPermissionsTable
(
  zbIeeeAddr_t aDevAddr,
  permissionsFlags_t  permissionsCategory
};
```

13.1.2   RemoveDeviceFromPermissionsTable

Removes a device from the Permission Configuration Table. This function only removes the device from the Permissions Configuration Table and not from the address map.

```
uint8_t RemoveDeviceFromPermissionsTable
(
  zbIeeeAddr_t  aDevAddr
)
```

13.1.3   RemoveAllFromPermissionsTable

Removes all Permission Configuration Table entries.

```
void RemoveAllFromPermissionsTable(void)
```

13.1.4   GetPermissionsTable

Obtains all the active entries in the Permission Configuration Table.

```
index_t GetPermissionsTable
(
  uint8_t * pDstTable
)
```

13.1.5   SetPermissionsTable

Sets an entry in the Permission Configuration Table.

```
void SetPermissionsTable
(
  index_t entryCounter,
  uint8_t * payload
)
```

Chapter 14
Frequency Agility

BeeStack supports an example implementation of a frequency agility channel manager which demonstrates how frequency agility could be implemented.

14.1 Frequency Agility Overview

The frequency agility module is described in Annex E of the *ZigBee Specification*, ZigBee Alliance, October 2007, and in Section 7.6.2 of the ZigBee 2007 specification. The following sections describe the example implementation in BeeStack. All source code is available in the `AppStackImpl.c`. All functions associated to Frequency Agility start with `FA_`.

The ZigBee specification defines one ZDP request and one ZDP indication used for frequency agility. the request mgmt_update_request requests a change of attributes like a channel mask, requests an energy scan or a channel change. The indication Mgmt_update_notify informs the system about transmission failures or results of an energy scan. See the ZigBee specification for further details.

NOTE

The ZigBee Specification refers to the frequency agility "channel" manager
in several ways; (Channel master, channel manager, etc.) From a ZDP
perspective, it is referred to as the nwk/network manager.

The main component of the Frequency Agility module is the channel manager. The channel manager makes the decision when to change the network to another channel. Several issues must be considered in the network manager:

- Zigbee 2007 and 2006 router detection
- Should the application decide when to change the channel
- How are energy scan reports analyzed
- Procedure to change the channel without leaving the network, for the RxOnWhenIdle = TRUE
- Router selection for Energy Scan Detection scan

BeeStack provides an example implementation of a frequency agility state machine.

14.1.1 Enabling the State Machine

The channel manager is optional and is enabled when frequency agility on the coordinator is enabled. It can also be present on another node by setting the nwk manager property.

NOTE

There can only be one channel manager in a network and it will typically be
the coordinator.

Every node is capable of being a channel manager if the following flags are enabled:

- gFrequencyAgilityCapability_d
- gEndDevCapability_d

14.1.2   Monitoring Indications

The channel manager monitors Mgmt_NWK_Update_notify() indications sent by other nodes in the network (see channel statistics). If the channel manager receives X (gMaxIncomingErrorReports_c) Mgmt_NWK_Update_notify indications in Y (gMaxTimeoutForIncomingErrorReports_c) amount of time, then an energy scan process is initiated.

The channel manager is found in `AppStackImpl.c`, in the FA_Process_Mgmt_NWK_Update_notify() function.

14.1.3   Energy Scan Process

The channel manager first does a local energy scan and then afterward requests Z(gMinNumberOfRouters_c) routers from the neighbor table to also do an energy scan. After waiting for the results of the energy scans or a time-out, the channel manager sends an error report to the application, and the application requests a "channel change", calling the FA_SelectChannelAndChange() function.

14.1.4   Application Control

All mgmt_update_notify indications and received mgmt_update_requests are sent to the ZDO callback handler so that the application can be in full control and the example implementation (FA_Process_Mgmt_NWK_Update_notify()) can be removed or changed by the application.

14.1.5   Transmission of a mgmt_update Notify

A node sends a mgmt_nwk_update_notify telegram every time its network layer reports a Tx Failure Report (NlmeTxReport) where there is X(gMaxTxFailuresPercentage_c) transmission failures. An NlmeTxreport is sent after Y transmissions (gMaxNumberOfTxAttempts_c).

If this occurs on the channel master itself, it does not send a mgmt_nwk_update_notify but directly starts the energy scan process.

14.1.6   Channel Change

When a channel change is requested, a mgmt_update_request is sent out on a broadcast address requesting the network to change channel, and the nodes will wait for a broadcast time-out before actually changing channels. In Freescale's implementation, the nodes will not do a nwk-rejoin but will simply change channels. A nwk-rejoin is not performed because routers would then "lose" their children.

14.1.7   Channel Statistics

When an error report is received, a channel change is initialized. The best channel is selected based on the statistics collected when receiving a mgmt_NWK_update_notify indication. An up/down averaging method is used for generating statistics. Each channel is represented with an 8 bit value which is initialized to a middle value of 0x7f. With each report, each channel is updated up or down based on the value for that channel in the report. The system uses the following ranges:

- 0xf0 - 0xff = +5
- 0xc0 - 0xef = +3
- 0xa0 - 0xbf = +1
- 0x7f - 0x9f = 0
- 0x50 - 0x7f = -1
- 0x25 - 0x4f = -3
- 0x00 - 0x24 = -5

The statistics are kept in a global (for averaging) and may be cleared with a ZDO reset.

14.2   Enabling Frequency Agility

Frequency agility example implementation is enabled by setting the gFrequencyAgilityCapability_d to TRUE in the Beestackconfiguration.h. In order for the implementation to work, the ZDP primitives must also be enabled. The gMgmt_NWK_Update_req_d and gMgmt_NWK_Update_notify_d options in the Beestackconfiguration.h file must also be set to TRUE.

By default, the coordinator is set as the Network manager. This can be changed with the gNetworkManagerCapability_d define. When this flag is set to true, then the node becomes a network manager. This is set in the server mask. This flag only affects coordinators and routers.

NOTE

The ZDP primitives can also be enabled without enabling the example implementation in case another implementation is needed.

14.3   Frequency Agility ZDP Primitives

To enable sending the ZDP commands Over the Air (OTA) from the application layer, it is necessary to send them through the ASL-ZDP interface. The following two functions are provided.

14.3.1   MGMT_NWK_Update_req

The ASL_Mgmt_NWK_Update_req is available when the flag gMgmtNwkUpdateRequest_d is True and it is not an End Device and gNwkManagerCapability_d =True.

Prototype

```
void ASL_Mgmt_NWK_Update_req(zbNwkAddr_t  aDestAddress, zbChannels_t aChannelList, uint8_t
iScanDuration, uintn8_t       iScanCount);
```

14.3.2   MGMT_NWK_Update_Notify

The ASL_Mgmt_Nwk_Update_Notify function is available when the flag gMgmtNwkUpdateNotify_d is True and it is not an End Device and gNwkManagerCapability_d =False.

Prototype

```
void ASL_Mgmt_NWK_Update_Notify( zbNwkAddr_t  aDestAddress, zbChannels_t aScannedChannels,
uint16_t       iTotalTransmissions, uint16_t       iTransmissionFailures, uint8_t
iScannedChannelListCount, zbEnergyValue_t  *paEnergyVslues, zbStatus_t    status);
```

14.4   Frequency Agility State Machine Primitives

The following section describes the Frequency Agility state machine primitives.

14.4.1   ASL_EnergyScanRequest

This function requests the lower layer to scan the channel.

Prototype

```
void ASL_EnergyScanRequest( zbChannels_t  aChannelList, uint8_t  duration);
```

14.4.2   ASL_ChangeChannel

This function requests the lower layer to change to a specific channel.

Prototype

```
void ASL_ChangeChannel(uint8_t  channelNumber);
```

14.4.3   FA_Process_Mgmt_NWK_Update_request

This function is available for any RxOnWhenIdle device, to process the incoming Mgmt_NWK_Update_Request (2.4.3.3.9 Mgmt_NWK_Update_req). Upon receipt, the Remote Device determines from the contents of the ScanDuration parameter whether this request is an update to the apsChannelMask and nwkManagerAddr parameters, a channel change command, or a request to scan channels and report the results.

Prototype

```
zbSize_t FA_Process_Mgmt_NWK_Update_request(zbMgmtNwkUpdateRequest_t *pMessageComingIn, void
*pMessageGoingOut, zbNwkAddr_t aSrcAddrr);
```

14.4.4   FA_Process_Mgmt_NWK_Update_notify

This function is only available for the Network Manager (Channel Master), to process the incoming Mgmt_NWK_Update_Notify. Upon receipt of an unsolicited Mgmt_NWK_Update_notify, the network manager must evaluate if a channel change is required in the network. For this there are several steps to follow:

Request other interference reports using the Mgmt_NWK_Update_req command. The network manager may request data from randomly selected routers in the network. (This is described in the Zigbee spec R17-Annex E, for the purpose of giving the user a simple example of how and where this is done and how it can be adjusted as needed).

Prototype

```
void FA_Process_Mgmt_NWK_Update_notify(zbMgmtNwkUpdateNotify_t  *pMessageComingIn);
```

14.4.5   FA_ChannelMasterStateMachine

The state machine used by the network manager to keep track of the FA procedure described in the ZigBee specification 053474r17 Annex E, uses a task and events. This state machine is also used by the routers participating in the FA procedure.

Prototype

```
void FA_ChannelMasterStateMachine(event_t events);
```

14.4.6   FA_ProcessEnergyScanCnf

The function is used either by the Network manager or by any other device participating on the FA process. This function catches every single energy scan confirm from the Nwk layer and processes them.

Prototype

```
void FA_ProcessEnergyScanCnf(nlmeEnergyScanCnf_t *pScanCnf  /* The Nlme scan confirm. */);
```

14.4.7   FA_ProcessNlmeTxReport

This function catches every time that the nwk layer sends a Tx report, to keep track of the transmission attempts and failures.

Prototype

```
void FA_ProcessNlmeTxReport(nlmeNwkTxReport_t  *pNlmeNwkTxReport );
```

Chapter 15
Interpan Communication

BeeStack supports the Interpan communication method specified by the Smart Energy/AMI application profile specification. The Interpan communication feature allows for communication outside the ZigBee network to very simple devices. For example, a refrigerator magnet may receive information about energy cost and display that information it using a multi-colored LED.

15.1 Interpan Communication Overview

The Interpan Communication feature is enabled by gInterPanCommunicationEnabled_c define in `Beestackconfiguration.h` to TRUE. Setting it to FALSE reduces the stack code size.

Interpan communication can also be enabled in the ZTC by setting the gSAPMessagesEnableInterPan_d define to TRUE.

NOTE
It is not possible to hook the Interpan SAPS.

BeeStack provides an interface for sending and receiving InterPan packets, AF_InterPanDataRequest() and InterPan_APP_SapHandler(). Detailed information on how to use the InterPan method can be found in the Smart energy/AMI application profile specification.

15.2 AF InterPan Data Request

The `AF_InterPanDataRequest()` function sends out an InterPan Data request out of the radio. Only 1 request pending is allowed.

The prototype is:

```
zbStatus_t AF_InterPanDataRequest( InterPanAddrInfo_t *pAddrInfo, uint8_t payloadLen, void
*pPayload, zbApsCounter_t  *pConfirmId);
```

The function can return the following status:

zbSuccess_c	If request created and sent. Confirm comes back to the application through the InterPan_APP_SapHandler().
gZbNoMem_c	Not enough memory to create the request. Wait and try again later.
gZbBusy_c	Message can not be sent (1 request is already pending.
gInvalidParameter_c	Invalid parameter supplied in address information

15.3   AF InterPan Data Indications

InterPan Data Indications arrives in the InterPan data indication sap handler, InterPan_APP_SapHandler().

Opposite AF Data Indications the InterPan Data indication messages must freed using the MSG_Free() instead of the AF_FreeDataIndicationMsg(). The InterPan_APP_SapHandler() can be found in `BeeAppInit.c`

Appendix A
Porting from BeeStack 2006 to BeeStack 2007

This appendix describes how to port from the ZigBee 2006 compliant version of BeeStack to the ZigBee 2007 compliant version of BeeStack.

Perform the following steps to port from a BeeStack 2006 solution to a BeeStack 2007 solution:

- Upgrade the BeeStack 2006 solution file to BeeStack 2007 solution file
- Consider enabling any new major features
- Apply changes required in the application

The following chapters describe each of these steps in detail.

A.1   Upgrading the Solution File

The BeeKit Wireless Connectivity Toolkit allows users to upgrade a solution file to a newer Codebase. The *BeeKit Wireless Connectivity Toolkit User's Guide* contains a detailed description of how to upgrade a solution file. Follow these steps to upgrade the solution file from BeeStack 2006 to BeeStack 2007.

NOTE

It is not possible to upgrade an HCS08 solution to an MC1322x (ARM7) solution. It is also not possible to upgrade from Codebase version 2.0.0 or or older to Codebase version 3.0.0 within BeeKit automatically. If upgrading from a Codebase version 2.0.0 or older, open the existing solution and record all the properties marked with **BOLD** (properties changed to a value different than their default value). Open the new Codebase, re-create the projects, apply the recorded changes and continue exporting the solution.

A.2   Enabling New Features

The BeeStack 2007 Codebase provides a range of new features that should be considered when porting:

NOTE

Enabling these new features impacts code size.

- Fragmentation - Allows sending larger packets than a standard MAC frame can contain by utilizing fragmentation.
- Frequency agility - Allows moving the network to a different channel in case a lot of noise is detected on an RF channel.
- ZDP primitives for Frequency agility - Enabling only the primitives allows the application developer to send the ZDP commands and then interpret them in the application.

- BeeStack CodeBase 3.0.0 and higher includes a new Device type (combo), new ZDO, link keys, ZigBee Pro Feature Set, Startup Attribute Set and Smart Energy application templates.

A.3   Applying Changes In The Application

A few changes in the application are needed to completely port an application from BeeStack 2006 to BeeStack 2007. The easiest porting method is to export the upgraded solution file and then compare the `beeapp.c` file in the work space with the one that the application developer has modified.

The following tasks must be performed:

- Modify the application NVM data set
- Use AF_FreeDataIndicationMsg() instead of MSG_Free() for freeing Data indications
- Insert code for handling received Frequency Agility messages

The following sections describe each of these tasks in detail.

A.3.1   Modifying The Application NVM Data Set

The gaNvAppDataSet structure has changed in the first line as follows:

From:

```
NvDataItemDescription_t const gaNvAppDataSet[] = {
  {&gBeeStackParameters,    sizeof(beeStackParameters_t)},  /* BeeStackSpecific */
............
```

To:

```
NvDataItemDescription_t const gaNvAppDataSet[] = {
  gAPS_DATA_SET_FOR_NVM,    /* APS layer NVM data */
............
```

A.3.2   Replacing MSG_Free() with AF_FreeDataIndicationMsg()

The BeeAppDataIndication() function receives the AF data indications. To make sure that all fragments in a fragmented data indication is freed the AF_FreeDataIndicationMsg() must be used instead.

Replace MSG_Free() with AF_FreeDataIndicationMsg() every place where a Data indication is freed.

NOTE

If the application is based on the WirelessUART, this must be done in RxZigBeeData().

BeeStack™ Software Reference Manual for ZigBee 2007, Rev. 1.1

A.3.3   Inserting Code for Handling Frequency Agility

Depending on which application template is used, either more, less, or no code must be inserted into the application. Doing a compare of the new and old `beeapp.c` file is the best way to determine what needs to be copied.

NOTE

If the application is built on the HA template, the changes are made in the ASL files and the `beeapp.c` does not need to be updated.

For the GenericApp, add the following code:

In BeeAppInit():

```
/* Get the statemachine for FA into start state. */
  FA_StateMachineInit();
}
```

Update the BeeAppZdpCallBack() to look as follows:

```
void BeeAppZdpCallBack
  (
  zdpToAppMessage_t *pMsg,
  zbCounter_t counter
  )
{
  uint8_t event;
  zbMatchDescriptorResponse_t * pMatchRsp;
  (void)counter;

  /* get the event from ZDP */
  event = pMsg->msgType;
  if(event == gzdo2AppEventInd_c) /* many possible application events */
    event = pMsg->msgData.zdo2AppEvent;

  /* got a response to match descriptor */
  switch(event) {

    case gMatch_Desc_rsp_c:
      pMatchRsp = &(pMsg->msgData.matchDescriptorResponse);
      if (pMatchRsp->status == gZbSuccess_c) {

        /* indicate matched */
        LED_SetLed(LED3, gLedOn_c);

        /* remember destination (nwkaddr + endpoint) */
        gAccelDstEndPoint = pMatchRsp->matchList[0];  /* match to first endpoint */
        Copy2Bytes(gaAccelDstAddr, pMatchRsp->aNwkAddrOfInterest);
        gfAccelFoundDst = TRUE;

        /* start sending the data */
        giAccelDemoState = accelStateStart_c;
        TS_SendEvent(gAppTaskID, accelEventState_c);
         }
```

BeeStack™ Software Reference Manual for ZigBee 2007, Rev. 1.1

```
          /* couldn't find match, give up */
            else
               LED_SetLed(LED3, gLedOff_c);
          break;

    /* network has been started */
    case gZDOToAppMgmtZCRunning_c:
    case gZDOToAppMgmtZRRunning_c:
    case gZDOToAppMgmtZEDRunning_c:
      if (appState == mStateIdle_c) {

        appState = mStateZDO_device_running_c;

        /* stop the flashing and indicate the device is running (has joined the network) */
        LED_SetLed(LED1, gLedOn_c);
        LCD_WriteString(1,(uint8_t *)"Running Device");
        }
      break;

    case gNlmeTxReport_c:
      FA_ProcessNlmeTxReport(&pMsg->msgData.nlmeNwkTxReport);
      break;

    case gMgmt_NWK_Update_notify_c:
      /* Already handle in ZDP. */
      break;

    case gNlmeEnergyScanConfirm_c:
      FA_ProcessEnergyScanCnf(&pMsg->msgData.energyScanConf);
      break;

    case gChannelMasterReport_c:
      FA_SelectChannelAndChange();
      break;
  }
  if (pMsg->msgType == gNlmeEnergyScanConfirm_c)
    MSG_Free(pMsg->msgData.energyScanConf.resList.pEnergyDetectList);

  /* free the message from ZDP */
  MSG_Free(pMsg);
}
```

A.3.4   New ZDO Features

BeeStack CodeBase 3.0.0 and above allows more control over how the ZDO initializes the device. Consider this extra control if any of these new features can enhance the application that is being ported. See Section 6.2, "General ZDO Interfaces (Codebase Version 3.0.0 and Higher)".

Appendix B
Table and Buffer Sizes

BeeStack contains several tables and buffers that can be adjusted to potentially free up memory or to enhance stack performance. This appendix describes the tables and buffers that impact stack size/performance and the effect that making adjustments to them will have on the system.

B.1   Message Buffer Configuration System

BeeStack uses and shares a pool of message buffers with the MAC layer to prevent heap fragmentation. Each buffer is a fixed size in an array. Over-the-air messages are called "big messages/buffers", and smaller messages, such as internal confirmations are called "small buffers/message". The RAM has a limited number of these buffers available. The `gTotalBigMsgs_d` and `gTotalSmallMsgs_d` defines in the `AppToMacPhyConfig.h` header file set the number of these buffers available in the system. The Freescale IEEE, 802.15.4 MAC documentation contains more details on the message buffer system. A Big buffer is about 150-159 bytes depending on the platform used and the stack configuration. Increasing the number of buffers, increases the number of messages that the node can contain. Decreasing the number of buffers, increases the chance of data starvation in dense networks.

Several properties in BeeStack have an indirect relation to the number of buffers and should be adjusted if the default buffer number is modified. These properties are as follows:

B.1.1   gHandleTrackingTableSize_c

This property is the maximum number of "Handle Tracking Table" entries which determines how many simultaneous messages from the higher layer the NWK layer can support. (That is, everything that ends up being a network layer message such as ZDP commands, APS commands and APS data.)

Range   1 - 255

Default   10

B.1.2   gPacketsOnHoldTableSize_c

This property is the maximum number of "Packets on Hold Table Size" entries which determines how many simultaneous messages the network layer can hold until they get routed. If this property is increased, the number of big buffers should also be increased.

Range   1 - 255

Default   2

B.1.3 gHttMaxIndirectEntries_c

This property is the maximum number of "Indirect Entries" held in the Handle Tracking table. Freescale recommends that the number of messages sent to the MAC for polling ZigBee End-Devices does not exceed the amount of big buffers minus three.

Range   1 - 255

Default   4

B.1.4 gApsMaxDataHandlingCapacity_c

Determines the number of simultaneous messages that the APS can handle from higher layers (both for the ZDP and the application). If this limit is reached, the system indicates busy (gZbBusy_c).

If increasing this value, users must also increase the number of timers in `TMR_Interface.h` with the same amount because one timer is required per message.

Range   1 - 7

Default   2

B.1.5 gDefaulEntriesInSKKEStateMachine_c

Determines the number of simultaneous SKKE processes can be handled at the same time (number of devices).

Range   One to as many big buffers that are available after the joining process.

Default   5

B.1.6 gDefaultEntriesInEAStateMachine_c

Determines the number of simultaneous "Entity Authentications" that can be handled at the same time (from different devices).

Range   One to as many big buffers that are available after the joining process.

Default   3

B.2 Address Map

The Address Map contains the mapping of the 64 bit IEEE addresses to 16 bit addresses. The address map is also used by the binding table and by the security system when link keys (on the trust center) are used because they both are required to map a 64 bit address to a 16 bit address. It is important that this table is big enough to contain an entry for each binding and each device that is given a link key.

The table entry size is 10 bytes, and should be increased depending on binding table entries and link key relations. The property name is gApsMaxAddrMapEntries_c and the default setting is nine (9).

NOTE

The address map is saved to NVM, so if its size is increased, verify that the table does not overflow the NVM page it resides on. Refer to the `nv_data.c` file to review NVM page information.

B.3   Binding Table

The Binding table contains the source bindings for the device it resides on. Each entry in the table contains a list of cluster IDs (services). If two devices bind multiple clusters on the same endpoint, only one binding table entry is required. This optimizes RAM consumption. For example, a dimmable switch and a light that typically would bind both the on/off and the level control cluster.

A binding table entry is 5 bytes + (2 * gMaximumApsBindingTableClusters_c).

The total number of binding table entries is set with the gMaximumApsBindingTableEntries_c property.

NOTE

The Binding table is saved to NVM, so if its size is increased, verify that the table does not overflow the NVM page it resides on. Refer to the `nv_data.c` file to review NVM page information.

B.4   Neighbor Table

The Neighbor table contains all the information of the child nodes and neighbors and is 15 bytes per entry for non-secure networks. The Neighbor table is adjusted using the properties option in BeeKit or in the `beestackconfiguration.h` file, depending on device type:

Router   gRouterNwkInfobaseMaxNeighborTableEntry_c

Coordinator   gCoordinatorNwkInfobaseMaxNeighborTableEntry_c

End device   gEndDevNwkInfobaseMaxNeighborTableEntry_c

If security is used, then additional memory is required to store a security frame counter for each device in the table. This is 10 bytes per neighbor table entry. Freescale recommends that the size of this table not be adjusted by the user.

NOTE

The Neighbor table is saved to NVM, so if its size is increased, verify that the table does not overflow the NVM page it resides on. Refer to the `nv_data.c` file to review NVM page information.

B.5   Link Key Table

The Link Key table contains the security material required for storing and maintaining a link key. If link keys are used, extra security material from this table is used and one entry per link key is supported. Link keys are enabled by setting the gApsLinkKeySecurity_d property to TRUE. The number of entries in the link key table/number of link keys supported is set using the gApsMaxLinkKeys_c property. The size of

the link key security material/link key table entry is 26 bytes. At least one address map entry per link key is required.

NOTE

The Link Key Table/security material is saved to NVM, so if its size is increased, verify that the table does not overflow the NVM page it resides on. Refer to the `nv_data.c` file to review NVM page information.

B.6   Routing and Route Discovery Tables

BeeStack has two Routing tables and one Route Discovery table. The regular Route table is adjusted using the gNwkInfobaseMaxRouteTableEntry_c property. Each entry is six bytes.

If the device is a concentrator (enabled with the gConcentratorFlag_d property, which is only valid on ZigBee PRO Feature set) the device will also contain a second Route table labeled as the Source Routing table. The Source Routing table is adjusted by the gNwkInfobaseMaxSourceRouteTableEntry_c property. Each entry is six bytes on HCS08 based transceivers like the MC1320x and MC1321x and eight bytes on ARM7 based transceivers like the MC1322x. The Route Discovery table is adjusted by the gNwkRoutingMaxRouteDiscoveyTableEntry_c property. Each entry is 29 bytes.

Freescale recommends not adjusting the Routing and Route Discovery tables to a value larger than what is required as a minimum by the Stack profile. In an interoperable network, users can not expect other vendor nodes to contain more than what the minimum requirements.

NOTE

None of the routing and route discovery tables are stored in NVM.