



Z-Stack

Application Programming Interface

Document Number: SWRA195

Texas Instruments, Inc.
San Diego, California USA

Version	Description	Date
1.0	Initial Release	12/11/2006
1.1	Added ZDO Device Network Startup	03/07/2007
1.2	Changes to ZDO interfaces	08/07/2007
1.3	Added Frequency Agility and PAN ID Conflict interfaces	12/18/2007
1.4	Added Many-To-One routing	01/23/2008
1.5	Added Inter-PAN transmission	04/09/2008
1.6	Added receive data description of afIncomingMSGPacket_t	03/19/2009
1.7	Updated APIs	03/30/2009
1.8	Fixed ZMacSetTransmitPower(), added ZMacLqiAdjustMode()	07/25/2010
1.9	Fixed StubAPS_RegisterApp() prototype description, updated Acronyms; added afAPSF_ConfigGet() & afAPSF_ConfigSet().	05/31/2011
1.10	Added AF_SUPPRESS_ROUTE_DISC_NETWORK to the AF_DataRequest() options field	10/19/2012
1.11	Update to include Z3.0 API. Added BDB subsection with its functions. Added GP subsection and its interface with the stubs.	11/15/2016
1.12	Adding bdb_StopInitiatorFindingBinding to BDB interface.	5/10/2017
1.13	Update bdb_setActiveCentralizedLinkKey to support fallback to default global centralized key and text plain key modes.	6/21/2018

Table of Contents

1. INTRODUCTION.....	1
1.1 PURPOSE.....	1
1.2 SCOPE.....	1
1.3 ACRONYMS.....	1
2. LAYER OVERVIEW.....	2
2.1 BDB.....	2
2.2 ZDO.....	2
2.3 AF.....	2
2.4 APS.....	2
2.5 NWK.....	2
2.6 GREEN POWER.....	2
2.7 ZMAC.....	2
3. APPLICATION PROGRAMMING INTERFACE.....	3
3.1 BASE DEVICE BEHAVIOR (BDB).....	3
3.1.1 Overview.....	3
3.1.2 BDB General Configuration and usage on a network.....	3
3.1.2.1 bdb_SetIdentifyActiveEndpoint ().....	3
3.1.2.2 bdb_setChannelAttribute ().....	4
3.1.2.3 bdb_RegisterIdentifyTimeChangeCB ().....	4
3.1.2.4 bdb_RegisterBindNotificationCB ().....	4
3.1.2.5 bdb_RegisterCommissioningStatusCB ().....	5
3.1.2.6 bdb_setCommissioningGroupID ().....	5
3.1.2.7 bdb_RepAddAttrCfgRecordDefaultToList ().....	6
3.1.2.8 bdb_RegisterForFilterNwkDesc ().....	6
3.1.2.9 bdb_TouchlinkSetAllowStealing ().....	7
3.1.2.10 bdb_RegisterTouchlinkTargetEnableCB ().....	7
3.1.3 BDB Security Configuration.....	7
3.1.3.1 bdb_GenerateInstallCodeCRC ().....	7
3.1.3.2 bdb_setJoinUsesInstallCodeKey ().....	8
3.1.3.3 bdb_setTCRequireKeyExchange ().....	8
3.1.3.4 bdb_addInstallCode ().....	8
3.1.3.5 bdb_RegisterTCLinkKeyExchangeProcessCB ().....	9
3.1.3.6 bdb_setActiveCentralizedLinkKey ().....	9
3.1.3.7 bdb_RegisterCBKETCLinkKeyExchangeCB ().....	10
3.1.4 BDB Application Runtime API.....	10
3.1.4.1 bdb_StartCommissioning ().....	10
3.1.4.2 bdb_initialize ().....	11

3.1.4.3	bdb_getZCLFrameCounter ().....	11
3.1.4.4	bdb_RepChangedAttrValue ().....	12
3.1.4.5	bdb_ZedAttemptRecoverNwk ().....	12
3.1.4.6	bdb_GetFBInitiatorStatus ().....	12
3.1.4.7	bdb_isDeviceNonFactoryNew ().....	13
3.1.4.8	bdb_nwkDescFree ().....	13
3.1.4.9	bdb_CBKETCLinkKeyExchangeAttempt ().....	13
3.1.4.10	touchLinkTarget_EnableCommissioning ().....	14
3.1.4.11	touchLinkTarget_DisableCommissioning ().....	14
3.1.4.12	touchLinkTarget_GetTimer ().....	15
3.1.4.13	bdb_TouchlinkGetAllowStealing ().....	15
3.1.4.14	bdb_resetLocalAction ().....	15
3.1.4.15	bdb_StopInitiatorFindingBinding ().....	16
3.1.5	<i>BDB Application Runtime Callbacks API</i>	16
3.1.5.1	bdbGCB_IdentifyTimeChange_t callback.....	16
3.1.5.2	bdbGCB_BindNotification_t callback.....	16
3.1.5.3	bdbGCB_CommissioningStatus_t callback.....	17
3.1.5.4	bdbGCB_CBKETCLinkKeyExchange_t callback.....	18
3.1.5.5	bdbGCB_TCLinkKeyExchangeProcess_t callback.....	19
3.1.5.6	bdbGCB_FilterNwkDesc_t callback.....	19
3.2	<i>ZIGBEE DEVICE OBJECTS (ZDO)</i>	20
3.2.1	<i>Overview</i>	20
3.2.2	<i>ZDO Device Network Startup</i>	20
3.2.2.1	ZDOInitDeviceEx().....	20
3.2.2.2	ZDOInitDevice().....	21
3.2.3	<i>ZDO Message Callbacks</i>	21
3.2.3.1	ZDO_RegisterForZDOMsg().....	21
3.2.3.2	ZDO_RemoveRegisteredCB().....	22
3.2.4	<i>ZDO Discovery API</i>	22
3.2.4.1	ZDP_NwkAddrReq().....	23
3.2.4.2	ZDP_NWKAddrRsp().....	24
3.2.4.3	ZDP_IEEEAddrReq().....	25
3.2.4.4	ZDP_IEEEAddrRsp().....	26
3.2.4.5	ZDP_NodeDescReq().....	27
3.2.4.6	ZDP_NodeDescMsg().....	27
3.2.4.7	ZDP_PowerDescReq().....	28
3.2.4.8	ZDP_PowerDescMsg().....	28

3.2.4.9	ZDP_SimpleDescReq()	29
3.2.4.10	ZDP_SimpleDescRsp()	29
3.2.4.11	ZDP_ComplexDescReq()	30
3.2.4.12	ZDP_ActiveEPIFReq ()	30
3.2.4.13	ZDP_ActiveEPIFRsp()	31
3.2.4.14	ZDP_MatchDescReq()	31
3.2.4.15	ZDP_MatchDescRsp()	32
3.2.4.16	ZDP_UserDescSet()	33
3.2.4.17	ZDP_UserDescConf()	33
3.2.4.18	ZDP_UserDescReq()	34
3.2.4.19	ZDP_UserDescRsp()	34
3.2.4.20	ZDP_DeviceAnnce()	35
3.2.4.21	ZDP_ServerDiscReq()	35
3.2.4.22	ZDP_ServerDiscRsp()	36
3.2.5	<i>ZDO Binding API</i>	37
3.2.5.1	ZDP_EndDeviceBindReq()	37
3.2.5.2	ZDP_EndDeviceBindRsp()	38
3.2.5.3	ZDP_BindReq()	38
3.2.5.4	ZDP_BindRsp()	39
3.2.5.5	ZDP_UnbindReq ()	39
3.2.5.6	ZDP_UnbindRsp()	40
3.2.6	<i>ZDO Management API</i>	41
3.2.6.1	ZDP_MgmtNwkDiscReq()	41
3.2.6.2	ZDP_MgmtNwkDiscRsp()	42
3.2.6.3	ZDP_MgmtLqiReq()	43
3.2.6.4	ZDP_MgmtLqiRsp()	43
3.2.6.5	ZDP_MgmtRtgReq ()	44
3.2.6.6	ZDP_MgmtRtgRsp()	44
3.2.6.7	ZDP_MgmtBindReq ()	45
3.2.6.8	ZDP_MgmtBindRsp()	45
3.2.6.9	ZDP_MgmtLeaveReq ()	46
3.2.6.10	ZDP_MgmtLeaveRsp()	47
3.2.6.11	ZDP_MgmtDirectJoinReq ()	47
3.2.6.12	ZDP_MgmtDirectJoinRsp()	48
3.2.6.13	ZDP_MgmtPermitJoinReq()	48
3.2.6.14	ZDP_MgmtPermitJoinRsp()	49
3.2.6.15	ZDP_MgmtNwkUpdateReq()	49

3.2.6.16	ZDP_MgmtNwkUpdateNotify()	50
3.2.7	<i>ZDO Parsing Functions</i>	50
3.2.7.1	ZDO_ParseAddrRsp	51
3.2.7.2	ZDO_ParseNodeDescRsp	51
3.2.7.3	ZDO_ParsePowerDescRsp	51
3.2.7.4	ZDO_ParseSimpleDescRsp	52
3.2.7.5	ZDO_ParseEPListRsp	52
3.2.7.6	ZDO_ParseBindRsp	52
3.2.7.7	ZDO_ParseMgmNwkDiscRsp	53
3.2.7.8	ZDO_ParseMgmtLqiRsp	53
3.2.7.9	ZDO_ParseMgmtRtgRsp	53
3.2.7.10	ZDO_ParseMgmtBindRsp	54
3.2.7.11	ZDO_ParseMgmtDirectJoinRsp	54
3.2.7.12	ZDO_ParseMgmtLeaveRsp	54
3.2.7.13	ZDO_ParseMgmtPermitJoinRsp	55
3.2.7.14	ZDO_ParseUserDescRsp	55
3.2.7.15	ZDO_ParseServerDiscRsp	55
3.2.7.16	ZDO_ParseEndDeviceBindReq	56
3.2.7.17	ZDO_ParseBindUnbindReq	56
3.2.7.18	ZDO_ParseUserDescConf	57
3.2.7.19	ZDO_ParseDeviceAnnce	57
3.2.7.20	ZDO_ParseMgmtNwkUpdateNotify	57
3.2.8	<i>ZDO Network Manager</i>	58
3.2.8.1	NwkMgr_SetNwkManager	58
3.3	APPLICATION FRAMEWORK (AF)	58
3.3.1	<i>Overview</i>	58
3.3.1.1	Endpoint Management	58
3.3.1.2	Sending Data	63
3.3.1.3	Receiving Data	65
3.4	APPLICATION SUPPORT SUB-LAYER (APS)	65
3.4.1	<i>Overview</i>	65
3.4.2	<i>Binding Table Management</i>	66
3.4.2.1	Binding Record Structure – BindingEntry_t	66
3.4.2.2	Binding Table Maintenance	66
3.4.2.3	Binding Table Searching	69
3.4.2.4	Binding Table Statistics	70
3.4.2.5	Binding Table Non-Volatile Storage	71

3.4.3	Group Table Management.....	71
3.4.3.1	Group Table Structures.....	72
3.4.3.2	Group Table Maintenance.....	72
3.4.3.3	Group Table Searching.....	73
3.4.3.4	Group Table Non-Volatile Storage.....	75
3.4.4	Quick Address Lookup.....	76
3.4.4.1	APSME_LookupExtAddr().....	76
3.4.4.2	APSME_LookupNwkAddr ().....	76
3.5	NETWORK LAYER (NWK).....	77
3.5.1	Network Management.....	77
3.5.1.1	NLME_NetworkDiscoveryRequest().....	77
3.5.1.2	NLME_NwkDiscReq2().....	78
3.5.1.3	NLME_NwkDiscTerm().....	79
3.5.1.4	NLME_NetworkFormationRequest().....	79
3.5.1.5	NLME_StartRouterRequest().....	80
3.5.1.6	NLME_JoinRequest().....	80
3.5.1.7	NLME_ReJoinRequest().....	81
3.5.1.8	NLME_OrphanJoinRequest().....	81
3.5.1.9	NLME_PermitJoiningRequest().....	82
3.5.1.10	NLME_DirectJoinRequest().....	82
3.5.1.11	NLME_LeaveReq().....	83
3.5.1.12	NLME_RemoveChild().....	83
3.5.1.13	NwkPollReq().....	84
3.5.1.14	NLME_SetPollRate().....	84
3.5.1.15	NLME_SetQueuedPollRate().....	84
3.5.1.16	NLME_SetResponseRate().....	85
3.5.1.17	NLME_RouteDiscoveryRequest().....	85
3.5.2	Address Management.....	86
3.5.2.1	Network Variables and Utility Functions.....	86
3.5.3	Network Non-Volatile Storage.....	90
3.5.3.1	NLME_UpdateNV().....	90
3.5.4	PAN ID Conflict.....	90
3.5.4.1	NLME_SendNetworkReport ().....	91
3.5.4.2	NLME_SendNetworkUpdate().....	91
3.5.5	Inter-PAN Transmission.....	92
3.5.5.1	StubAPS_SetInterPanChannel().....	92
3.5.5.2	StubAPS_SetIntraPanChannel().....	92

3.5.5.3	StubAPS_InterPan()	92
3.5.5.4	StubAPS_RegisterApp()	93
3.6	GREEN POWER LAYER	93
3.6.1	GP_DataInd()	93
3.6.2	GP_DataReq()	93
3.6.3	GP_DataCnf()	94
3.6.4	GP_SecReq()	94
3.6.5	GP_SecRsp()	94
3.7	ZMAC LAYER (ZMAC)	95
3.7.1	ZMacSetTransmitPower()	95
3.7.2	ZMacLqiAdjustMode()	95

1. Introduction

1.1 Purpose

This document provides the Application Programmers Interface (API) for the Z-Stack™ components provided as part of the ZigBee-2015 compliant Z-Stack release.

1.2 Scope

This document enumerates the implemented API of all components provided as part of the ZigBee-2015 compliant Z-Stack release. Details pertinent to each interface, including data structures and function calls are specified in sufficient detail so as to allow a programmer to understand and utilize them during development. API's are presented from the top (Application) level down.

1.3 Acronyms

AF	Application Framework
AIB	APS Information Base
API	Application Programming Interface
APS	Application Support Sub-Layer
APSDE	APS Date Entity
APSME	APS Management Entity
ASDU	APS Service Datagram Unit
BDB	Base Device Behavior
GP	Green Power
LQI	Link Quality Indicator
MAC	Media Access Control
MSG	Message
NHLE	Next Higher Layer Entity
NIB	Network Information Base
NWK	Network
NV	Non-Volatile memory
PAN	Personal Area Network
STAR	A network topology consisting of one master device and multiple slave devices
ZDO	ZigBee Device Object

2. Layer Overview

This section provides an overview of the layers covered in this document.

2.1 BDB

The Base Device Behavior layer specifies the environment, initialization, commissioning and operating procedures of a base device operating on the ZigBee-PRO stack to ensure profile interoperability. The Z-Stack BDB provides the data structures and interface that an application developer requires to comply with this specification. It also provides a generic implementation of reporting attributes for applications that supports clusters with reportable attributes.

2.2 ZDO

The ZigBee Device Objects (ZDO) layer provides functionality for managing a ZigBee device. The ZDO API provides the interface for application endpoints to manage functionality for ZigBee Coordinators, Routers, or End Devices which includes creating, discovering, and joining a ZigBee network; binding application endpoints; and managing security.

2.3 AF

The Application Framework (AF) interface supports an Endpoints (including the ZDO) interface to the underlying stack. The Z-Stack AF provides the data structures and helper functions that a developer requires to build a device description and is the endpoint multiplexor for incoming messages.

2.4 APS

The Application Support Sublayer (APS) API provides a general set of support services that are used by both the ZDO and the manufacturer-defined application objects.

2.5 NWK

The ZigBee network (NWK) layer provides management and data services to higher layer (application) components.

2.6 Green Power

The Green Power layer data services that allows the higher layer to send or receive data from the Green Power stubs. These stubs are used to interact with green power devices either for commissioning or normal operation in the network.

2.7 ZMAC

The ZMAC layer provides an interface between the 802.15.4 MAC and the ZigBee NWK layers.

3. Application Programming Interface

This section provides a summary of the commonly used data structures implemented in the Z-Stack, as well as the API for accessing key functionality provided by the specified layer.

3.1 Base Device Behavior (BDB)

This section enumerates all the function calls provided by the BDB layer that are necessary for the implementation of all commands and responses defined in Base Device Behavior (BDB), as well as other functions to ensure interoperability between different ZigBee Devices. All BDB API functions are categorized by general and security configuration functions and by function that can be call by the application at runtime. Each category is discussed in the following sections.

3.1.1 Overview

BDB describes how a general ZigBee Device will enter and behave in the ZigBee network ensuring profile interoperability between devices. More specifically, the BDB specification defines the following functionalities:

- The initialization procedures
- The commissioning procedures
- The reset procedures
- The security procedures
- Reporting attributes

All these procedures are performed by using the existing API from other layers such as ZDO, ZCL, NWK and APS.

3.1.2 BDB General Configuration and usage on a network

All of the BDB API is contained in a single header file, the `bdb_interface.h`. Based on the BDB functionalities and the methods defined in the header file, this API functions are presented in the following subsections:

- Functions that the application must use to configure BDB parameters.
- Functions use by the application to configure BDB security- specific parameters.
- Functions called by the applications after starting the BDB Commission and during the runtime of the application.
- Callbacks functions provided by the BDB to notify events or execute custom application code in certain parts of the BDB Commissioning procedure.

3.1.2.1 `bdb_SetIdentifyActiveEndpoint ()`

Set the endpoint which will perform the finding and binding (either Target or Initiator).

Prototype

```
ZStatus_t bdb_SetIdentifyActiveEndpoint( uint8 activeEndpoint );
```

Parameter Details

`activeEndpoint` – active endpoint with which perform Finding and Binding. If set to 0xFF all application endpoints with Identify will be attempted.

Return

ZStatus_t – status values defined in ZStatus_t in ZComDef.h.

3.1.2.2 bdb_setChannelAttribute ()

Set the primary or secondary channel for discovery or formation procedure. By default the primary channel is set to DEFAULT_CHANLIST and the secondary channel is set to (DEFAULT_CHANLIST ^ 0x07FFF800).

Prototype

```
void bdb_setChannelAttribute( bool isPrimaryChannel, uint32 channel );
```

Parameter Details

isPrimaryChannel – True if channel to set is primary, False if the channel to set is the secondary.

channel – bit mask containing the channels to scan for this request. The channel definitions are contained in f8wConfig.cfg

Return

None.

3.1.2.3 bdb_RegisterIdentifyTimeChangeCB ()

Register an Application's Identify Time change callback function to let know the application when identify is active or not.

Prototype

```
void bdb_RegisterIdentifyTimeChangeCB( bdbGCB_IdentifyTimeChange_t pfnIdentifyTimeChange );
```

Parameter Details

pfnIdentifyTimeChange – application callback function to let know the application when identify is active or not.

Return

None.

3.1.2.4 bdb_RegisterBindNotificationCB ()

Register an Application's notification callback function to let know the application when a new bind is added to the binding table. This callback will include the address of the remote device, the application endpoint and the cluster which have created the bind.

Prototype

```
void bdb_RegisterBindNotificationCB( bdbGCB_BindNotification_t pfnBindNotification );
```

Parameter Details

`pfnBindNotification` – application callback function to let know the application when a new bind is added to the binding table.

Return

None.

3.1.2.5 bdb_RegisterCommissioningStatusCB ()

Register a callback in which the status of the procedures done in BDB commissioning process will be reported. It also reports the remaining commissioning modes to be attempted.

Prototype

```
void bdb_RegisterCommissioningStatusCB( bdbGCB_CommissioningStatus_t bdbGCB_CommissioningStatus );
```

Parameter Details

`bdbGCB_CommissioningStatus` – application callback function in which the status of the BDB commissioning process will be reported.

Return

None.

3.1.2.6 bdb_setCommissioningGroupID ()

Set the group Id which matching clusters will be added by subsequent calls to Finding & Binding commissioning method as initiator.

Prototype

```
void bdb_setCommissioningGroupID( uint16 groupID );
```

Parameter Details

`groupID` – Commissioning Group ID to which matching clusters will be added. If set to 0xFFFF then no groups are used by Finding & Binding. The default Commissioning Group ID is 0xFFFF.

Return

None.

3.1.2.7 bdb_RepAddAttrCfgRecordDefaultToList ()

Adds default configuration values for a Reportable Attribute. This configuration will be used when a bind is created for the attribute in the specified pair Cluster-Endpoint. If no configuration is defined for a reportable attribute, then the default configuration will be used, which can be defined in `bdb_interface.h`. Stack default is set that attribute as no periodic reporting.

Prototype

```
ZStatus_t bdb_RepAddAttrCfgRecordDefaultToList( uint8 endpoint, uint16 cluster, uint16 attrID,
                                                uint16 minReportInt, uint16 maxReportInt, uint8* reportableChange );
```

Parameter Details

`endpoint` – the endpoint ID of the Reportable Attribute Record to configure it's defaults values.

`cluster` – the cluster ID of the Reportable Attribute Record to configure it's defaults values.

`attrID` – the attribute ID of the Reportable Attribute Record to configure it's defaults values.

`minReportInt` – the default value for minimum reportable interval of the Reportable Attribute Record.

`maxReportInt` – the default value for maximum reportable interval of the Reportable Attribute Record.

`reportableChange` – buffer containing the attribute value that will be the delta change needed to trigger a report.

Return

`ZStatus_t` – returns `ZInvalidParameter` if the given endpoint, cluster, attribute ids tuple was not found in the applications defined simple descriptors. `ZFailure` if there was no memory to allocate the entry. `ZSuccess` if the default entry was saved successfully.

3.1.2.8 bdb_RegisterForFilterNwkDesc ()

Register a callback in which the application gets the list of network descriptors got from active scan. Use `bdb_nwkDescFree` to release the network descriptors that are not of interest and leave those which are to be attempted to join. Example filter networks which extended PAN ID do not match the manufacturer.

Prototype

```
void bdb_RegisterForFilterNwkDesc( bdbGCB_FilterNwkDesc_t bdbGCB_FilterNwkDesc );
```

Parameter Details

`bdbGCB_FilterNwkDesc` – application callback function in which the application gets the list of network descriptors got from active scan.

Return

None.

3.1.2.9 bdb_TouchlinkSetAllowStealing ()

General function to allow stealing when performing TL as target.

Prototype

```
void bdb_TouchlinkSetAllowStealing( bool allow );
```

Parameter Details

`allow` – allow stealing if TRUE, deny if FALSE.

Return

None.

3.1.2.10 bdb_RegisterTouchlinkTargetEnableCB ()

Register an Application's Enable/Disable callback function. Refer to `touchLinkTarget_EnableCommissioning` to enable/disable TL as target.

Prototype

```
void bdb_RegisterTouchlinkTargetEnableCB( tIGCB_TargetEnable_t pfnTargetEnableChange );
```

Parameter Details

`pfnTargetEnableChange` – application callback function which is called when the Touchlink process is enabled or disabled in the target.

Return

None.

3.1.3 BDB Security Configuration

In this section is defined the API to configure the security parameters to be used by joining devices and the Trust Center, such set install codes, Trust Center Policies or validate Install Codes.

3.1.3.1 bdb_GenerateInstallCodeCRC ()

This is a utility function that calculates the CRC value for the install code passed. This is meant to be used for debug purposes and/or validate the CRC introduced to devices by manufacturer specific way.

Prototype

```
uint16 bdb_GenerateInstallCodeCRC( uint8 *installCode );
```

Parameter Details

installCode –buffer different from NULL with length of 16 bytes containing the install code from which CRC will be generated.

Return

uint16 CRC –CRC generated from the install code introduced.

3.1.3.2 bdb_setJoinUsesInstallCodeKey ()

This function sets the BDB attribute bdbJoinUsesInstallCodeKey defined in the BDB specification. This is only for Trust Center devices.

Prototype

```
void bdb_setJoinUsesInstallCodeKey( bool set );
```

Parameter Details

set – If TRUE only devices with the install code register in the trust center can join the network, otherwise devices may or not have an install code register.

Return

None.

3.1.3.3 bdb_setTCRequireKeyExchange ()

Set the bdb_setTCRequireKeyExchange attribute, this setting may be use to control if legacy devices (prior Zigbee PRO R20) are allowed to join in the network. This is only for Trust Center devices.

Prototype

```
void bdb_setTCRequireKeyExchange( bool isKeyExchangeRequired );
```

Parameter Details

isKeyExchangeRequired – True if Trust Center will remove devices that do not perform key exchange after bdbTrustCenterNodeJoinTimeout, False to not remove devices.

Return

None.

3.1.3.4 bdb_addInstallCode ()

This function allows the Trust Center to add a preconfigured key for a specific joining device. The format of the key is defined by the compilation flag BDB_INSTALL_CODE_USE, which defines if the install code is 16 bytes install code plus 2 bytes CRC or 16 bytes of plain text key. This is only for Trust Center devices, joining devices must refer to bdb_setActiveCentralizedLinkKey() in section 3.1.3.6

Prototype

```
ZStatus_t bdb_addInstallCode( uint8* pInstallCode, uint8* pExt );
```

Parameter Details

pInstallCode – buffer array containing the Install Code in the format of Install code or plain text key.

pExt – Extended address of the device associated with the given Install Code.

Return

ZStatus_t – status values defined in ZStatus_t in ZComDef.h. ZInvalidParameter if pInstallCode or pExt is NULL or the given CRC is incorrect, Zsuccess if the APS TC Link key was correctly added. ZApsTableFull is the APS TC Link key is full.

3.1.3.5 bdb_RegisterTCLinkKeyExchangeProcessCB ()

Register a callback to receive notifications on the joining devices and its status on TC link key exchange.

Prototype

```
void bdb_RegisterTCLinkKeyExchangeProcessCB( bdbGCB_TCLinkKeyExchangeProcess_t  
                                             bdbGCB_TCLinkKeyExchangeProcess );
```

Parameter Details

bdbGCB_TCLinkKeyExchangeProcess – application callback function to receive notifications on the joining devices and its status on TC link key exchange.

Return

None.

3.1.3.6 bdb_setActiveCentralizedLinkKey ()

This function sets the active centralized key to be used, Global or Install Code derived. The format of the key is defined by the compilation flag BDB_INSTALL_CODE_USE, which defines if the install code is 16 bytes install code plus 2 bytes CRC or 16 bytes of plain text key. This function is only for joining devices.

Prototype

```
ZStatus_t bdb_setActiveCentralizedLinkKey( uint8 zstack_CentralizedLinkKeyModes, uint8* pBuf );
```

Parameter Details

useGlobal – If TRUE, then use default TC link key, if FALSE use pBuf as source for IC key derived input.

pBuf – buffer array containing the Install Code in the format of Install code or plain text key.

Return

`ZStatus_t` – ZFailure when no valid `BDB_INSTALL_CODE_USE` is used, `ZInvalidParameter` when Install Code buffer is null.

3.1.3.7 bdb_RegisterCBKETCLinkKeyExchangeCB ()

Register a callback in which the TC link key exchange procedure will be performed by application. The result from this operation must be notified by using the `bdb_CBKETCLinkKeyExchangeAttempt` interface. The normal Trust Center Link Key Exchange procedure will be performed if no callback is registered or if the TC link key exchange procedure is reported as failed.

Prototype

```
void bdb_RegisterCBKETCLinkKeyExchangeCB( bdbGCB_CBKETCLinkKeyExchange_t
bdbGCB_CBKETCLinkKeyExchange );
```

Parameter Details

`bdbGCB_CBKETCLinkKeyExchange` – application callback function to perform the TC link key exchange.

Return

None.

3.1.4 BDB Application Runtime API

In this section is described the API to be used when the devices are already configured and will start interacting in the network by being commissioned or decommissioned of the network.

3.1.4.1 bdb_StartCommissioning ()

Initialize the device with persistent data if this was not done already, the initialization process restores the network (Silent rejoin for ZC and ZR, Rejoin for ZED is attempted), retrieve binds and resume reporting attributes if any. It also starts the commissioning process with the commissioning mode mask given. If another commissioning mode is already in process, the commissioning mode mask is appended to the current tasks. The commissioning mode mask is process with the following priority. *Touchlink*, *Network Steering*, *Formation* and *Finding and Binding*. The commissioning modes priority to be executed are validated when a commissioning mode is finish, so if a commissioning mode with higher priority is appended, it will be executed before a commissioning mode with lower priority even if it has been appended before. Refer to section **!Error! No se encuentra el origen de la referencia.** on how to receive notifications from the commissioning modes being executed.

Prototype

```
void bdb_StartCommissioning( uint8 mode );
```

Parameter Details

`mode` – mask with the commissioning modes to be executed.

Macro	Value	Description
-------	-------	-------------

BDB_COMMISSIONING_MODE_INITIATOR_TL	0x01	Request touchlink as initiator
BDB_COMMISSIONING_MODE_NWK_STEERING	0x02	Request network steering
BDB_COMMISSIONING_MODE_NWK_FORMATION	0x04	Request network formation
BDB_COMMISSIONING_MODE_FINDING_BINDING	0x08	Request finding and binding

Return

None.

3.1.4.2 bdb_initialize ()

Initialize the device with persistent data if this was not done already, the initialization process restores the network (Silent rejoin for ZC and ZR, Rejoin for ZED is attempted), retrieve binds and resume reporting attributes if any.

Prototype

```
#define bdb_initialize() bdb_StartCommissioning(BDB_COMMISSIONING_INITIALIZATION)
```

Parameter Details

None.

Return

None.

3.1.4.3 bdb_getZCLFrameCounter ()

Get the next sequence number for ZCL frames. This function first increments the sequence number then returns the resulted value and is based on bdb_ZclTransactionSequenceNumber, which is used by the stack to keep track of the ZCL sequence number.

Prototype

```
uint8 bdb_getZCLFrameCounter( void );
```

Parameter Details

None.

Return

uint8 – next sequence number to use in a ZCL packet.

3.1.4.4 bdb_RepChangedAttrValue ()

Notify BDB reporting attribute module about the change of an attribute value to validate the triggering of a reporting attribute message. This function **MUST** be called whenever a reportable attribute value is changed in order to generate the Reporting Attribute message if the conditions specified by the ZCL specification are met.

Prototype

```
ZStatus_t bdb_RepChangedAttrValue( uint8 endpoint, uint16 cluster, uint16 attrID );
```

Parameter Details

endpoint – the endpoint ID of the Reportable Attribute that has a change in its value.

cluster – the cluster ID of the Reportable Attribute that has a change in its value.

attrID – the attribute ID of the Reportable Attribute that has a change in its value.

Return

ZStatus_t – ZInvalidParameter - No endpoint, cluster, attribute ID found in simple descriptor. ZSuccess is return is the attribute change was notified successfully.

3.1.4.5 bdb_ZedAttemptRecoverNwk ()

This function allows End Devices to rejoin its commissioned network after losing its parent or being power cycled and no parent device were found by the initialization process. This should be called periodically if the device does not found any suitable parent device until it joins the commissioned network or this devices gets commissioned into another network by performing a factory new reset.

Prototype

```
uint8 bdb_ZedAttemptRecoverNwk( void );
```

Parameter Details

None.

Return

ZStatus_t – ZSuccess if the attempt is being executed. ZFailure if device do not have network parameters to perform this action.

3.1.4.6 bdb_GetFBIInitiatorStatus ()

Get the F&B initiator status for periodic requests. Refer to the configuration macros FINDING_AND_BINDING_PERIODIC_ENABLE and FINDING_AND_BINDING_PERIODIC_TIME in bdb_interface.h

Prototype

```
void bdb_GetFBInitiatorStatus( uint8 *RemainingTime, uint8* AttemptsLeft );
```

Parameter Details

RemainingTime – pointer in which is passed the time left in seconds.

AttemptsLeft – pointer in which the number of attempts left are passed.

Return

None.

3.1.4.7 bdb_isDeviceNonFactoryNew ()

Returns the state of `bdbNodeIsOnANetwork` attribute in the BDB ZStack. This attribute indicates if the device is commissioned on a network or not.

Prototype

```
bool bdb_isDeviceNonFactoryNew( void );
```

Parameter Details

None.

Return

`bool` – True if the device is factory new. False otherwise.

3.1.4.8 bdb_nwkDescFree ()

This function frees a network descriptor obtained from the `bdbGCB_FilterNwkDesc_t` callback function. This can be used to filter networks during the commissioning process.

Prototype

```
ZStatus_t bdb_nwkDescFree( networkDesc_t* nodeDescToRemove );
```

Parameter Details

`nodeDescToRemove` – Pointer to the node descriptor that will be freed.

Return

`ZStatus_t` – `ZSuccess` - If the device was found and erased. `ZInvalidParameter` is pointer was not found.

3.1.4.9 bdb_CBKETCLinkKeyExchangeAttempt ()

This function must be called by the application to tell BDB module that the application defined Trust Center Link exchange has finished and inform the result. If the application defined mechanism to perform the Trust Center Link

Key exchange fails BDB will try the default Trust Center Link exchange process, if the application success it will keep going with the joining process. If no application defined Trust Center Link exchange was registered in the configuration phase then the application can omit calling this function at runtime.

Prototype

```
void bdb_CBKETCLinkKeyExchangeAttempt( bool didSuccess );
```

Parameter Details

`didSuccess` – True if the application defined Trust Center Link exchange process was a success, False otherwise.

Return

None.

3.1.4.10 touchLinkTarget_EnableCommissioning ()

Enable the reception of TouchLink commissioning commands for a specific time. Refer to `bdb_RegisterTouchlinkTargetEnableCB` to get enable/disable notifications. If timeout time is equal or greater than `TOUCHLINK_TARGET_PERPETUAL`, then touchlink as target is enabled indefinitely.

Prototype

```
void touchLinkTarget_EnableCommissioning( uint32 timeoutTime );
```

Parameter Details

`timeoutTime` – Enable timeout in milliseconds.

Return

None.

3.1.4.11 touchLinkTarget_DisableCommissioning ()

Disable the TouchLink commissioning on a target device. Refer to `bdb_RegisterTouchlinkTargetEnableCB` to get enable/disable notifications.

Prototype

```
void touchLinkTarget_DisableCommissioning( void );
```

Parameter Details

None.

Return

None.

3.1.4.12 touchLinkTarget_GetTimer ()

Get the remaining time for TouchLink as target.

Prototype

```
uint32 touchLinkTarget_GetTimer( void );
```

Parameter Details

None.

Return

uint32 – remaining time of the currently enabled TouchLink commissioning. Units in milliseconds.

3.1.4.13 bdb_TouchlinkGetAllowStealing ()

General function to get *AllowStealing* configuration parameter when performing TouchLink as target.

Prototype

```
bool bdb_TouchlinkGetAllowStealing( void );
```

Parameter Details

None.

Return

bool – return TRUE if allowed, FALSE if not allowed.

3.1.4.14 bdb_resetLocalAction ()

Application interface to perform BDB reset to factory new. This will cause the device to clear all Zigbee persistent data, except the outgoing network frame counter associated to the commissioned network.

Prototype

```
void bdb_resetLocalAction( void );
```

Parameter Details

None.

Return

None.

3.1.4.15 bdb_StopInitiatorFindingBinding ()

Stops Finding&Binding for initiator devices. The result of this process is reported in bdb notifications callback.

Prototype

```
void bdb_StopInitiatorFindingBinding ( void );
```

Parameter Details

None.

Return

None.

3.1.5 BDB Application Runtime Callbacks API

The BDB to application communication is handled by the BDB callbacks; these callbacks usually provide notifications on the process managed by BDB or enable the application to execute custom code.

3.1.5.1 bdbGCB_IdentifyTimeChange_t callback

This callback will be trigger by the BDB when the Identify mode starts by any means, then every second until the identify time has ended for the given endpoint. The application must register a callback method using `bdb_RegisterIdentifyTimeChangeCB` in the configuration phase in order to get these calls.

Prototype

```
typedef void (*bdbGCB_IdentifyTimeChange_t)( uint8 endpoint );
```

Parameter Details

`endpoint` – the reported Identify mode status is related to this endpoint ID.

Return

None.

3.1.5.2 bdbGCB_BindNotification_t callback

This callback is triggered by the BDB to inform the application when a new bind is added to the Binding Table by any mean. The application must register a callback method using `bdb_RegisterBindNotificationCB` in the configuration phase in order to get these calls.

Prototype

```
typedef void (*bdbGCB_BindNotification_t)( bdbBindNotificationData_t *bindData );
```


Parameter Details

`bindData` – pointer to a structure that contains the information of the bind that was recently added.

Return

None.

3.1.5.3 bdbGCB_CommissioningStatus_t callback

This callback is triggered by the BDB to inform the application the status of BDB Commissioning process when started. The application must register a callback method using `bdb_RegisterCommissioningStatusCB` in the configuration phase in order to get these calls.

Prototype

```
typedef void (*bdbGCB_CommissioningStatus_t)(bdbCommissioningModeMsg_t *bdbCommissioningModeMsg);
```

Parameter Details

`bdbCommissioningModeMsg` – pointer to a structure (defined in `bdb_interface.h`) that contains the information related to the current status of the BDB Commissioning. This structure contains the remaining commissioning modes to execute, the commissioning mode process status notification and the status.

Commissioning modes reported and its status

Commissioning mode (BDB_COMMISSIONING_mode)	Status reported (BDB_COMMISSIONING_status)	Description
INITIALIZATION	NETWORK_RESTORED	Only send if the device did restore its network parameters. On end devices, if no parent is found with the restored network parameters, a <i>Parent Lost</i> mode is with status <i>No Network</i> is notified.
NWK_STEERING (for Router and End Devices)	IN_PROGRESS	Notifies when network steering is started (only if the device is not in a network, otherwise reports <i>success</i>)
	NO_NETWORK	No suitable network was found in primary channel or secondary channel masks or the joining process did fail in the attempted networks.
	TCLK_EX_FAILURE	The device successfully joined the network, but could not perform the Trust Center Link Key exchange process. The device will reset to factory new after this notification is reported to the application.
	SUCCESS	The device is now on a network and broadcasted a Management Permit Joining ZDO frame.

NWK_STEERING (for Coordinators)	NO_NETWORK	The device is not on a network, so it cannot perform this action.
	SUCCESS	The device is in a network and has broadcasted a Management Permit Joining ZDO frame.
FORMATION	IN_PROGRESS	Notifies when formation process is started.
	SUCCESS	The network has been created successfully.
	FORMATION_FAILURE	The device could not create the network with the given parameters.
FINDING_BINDING	FB_TARGET_IN_PROGRESS	Indicates the start of the Finding and Binding as target. No notification is given by this callback when the process ends. For the notification when this ends refer to 3.1.5.1
	FB_INITIATOR_IN_PROGRESS	Indicates the start of the Finding and Binding as Initiator.
	FB_NO_IDENTIFY_QUERY_RESPONSE	After complete the Finding and Binding process as initiator (single attempt of periodic attempt), no identify query responses were received.
	FB_BINDING_TABLE_FULL	During the Finding and Binding process the binding table got full, so the process stops and no additional binds can be added.
	FAILURE	No endpoint was found to perform Finding and Binding, or the endpoint did not have implemented the Identify cluster properly.
TOUCHLINK	TL_TARGET_FAILURE	A node has not joined a network when requested during touchlink.
	TL_NOT_AA_CAPABLE	The initiator is not address assignment capable during touchlink
	TL_NO_SCAN_RESPONSE	No response to a Scan Reques inter-PAN command has been received during touchlink
	TL_NOT_PERMITTED	A touchlink steal attempt was made when a node is already connected to a centralized security network.
PARENT_LOST (Only for End Devices)	NO_NETWORK	This is notified if the end device does lose contact with the parent device or if after initialization it cannot find a parent device in the commissioned network.
	NETWORK_RESTORED	Notification that a suitable parent device got found and the rejoin process was successful.

Return

None.

3.1.5.4 bdbGCB_CBKETCLinkKeyExchange_t callback

This callback is called by the BDB when the device has joined the network and it's ready to perform the application defined Trust Center Link Key exchange which is started or contained inside this function. The application must register a callback method using `bdb_RegisterCBKETCLinkKeyExchangeCB` in the configuration phase in order for the BDB to execute this application specific link exchange. After the application has finished its Trust Center Link Key exchange it must called the function `bdb_CBKETCLinkKeyExchangeAttempt` so that the BDB can resume Commissioning.

Prototype

```
typedef void (*bdbGCB_CBKETCLinkKeyExchange_t)( void );
```

Parameter Details

None.

Return

None.

3.1.5.5 bdbGCB_TCLinkKeyExchangeProcess_t callback

This callback is triggered by the BDB to notify the application about the joining devices and its status on the Trust Center Link Key exchange. The application must register a callback method using `bdb_RegisterTCLinkKeyExchangeProcessCB` in the configuration phase in order to get these calls.

Prototype

```
typedef void (*bdbGCB_TCLinkKeyExchangeProcess_t) (bdb_TCLinkKeyExchProcess_t*  
                                                    bdb_TCLinkKeyExchProcess);
```

Parameter Details

`bdb_TCLinkKeyExchProcess` – pointer to a structure (defined in `bdb_interface.h`) that contains the information related to the status of the Trust Center Key Exchange joined device.

Return

None.

3.1.5.6 bdbGCB_FilterNwkDesc_t callback

This callback is used by the BDB to inform the application of the result of doing an active scan; the resulted information is in the form of network descriptors. Use `bdb_nwkDescFree` to release the network descriptors that are not of interest and leave those which are to be attempted. The application must register a callback method using `bdb_RegisterForFilterNwkDesc` in the configuration phase in order to get these calls.

Prototype

```
typedef void (*bdbGCB_FilterNwkDesc_t) (networkDesc_t *pBDBListNwk, uint8 count);
```

Parameter Details

`pBDBListNwk` – pointer to a structure (defined in `NLMEDE.h`) that contains the information of a network descriptor of a network found in the active scan.

`count` – number of network descriptors in the list `pBDBListNwk`.

Return

None.

3.2 ZigBee Device Objects (ZDO)

This section enumerates all the function calls provided by the ZDO layer that are necessary for the implementation of all commands and responses defined in ZigBee Device Profile (ZDP), as well as other functions that enable devices to operate as ZigBee Devices. All ZDO API functions are categorized by their functionalities in the overview section. Each category is discussed in the following sections.

3.2.1 Overview

ZDP describes how general ZigBee Device features are implemented within ZDO. It defines Device Description and Clusters which employ command and response pairs. Through the definition of messages in command structure, ZDP provides the following functionality to the ZDO and applications.

- Device Network Startup
- Device and Service Discovery
- End Device Bind, Bind and Unbind Service
- Network Management Service

Device discovery is the process for a ZigBee device to discover other ZigBee Devices. One example of device discovery is the NWK address request which is broadcast and carries the known IEEE address as data payload. The device of interest should respond and inform its NWK address. Service discovery provides the ability for a device to discover the services offered by other ZigBee devices on the PAN. It utilizes various descriptors to specify device capabilities.

End device bind, bind and unbind services offer ZigBee devices the binding capabilities. Typically, binding is used during the installation of a network when the user needs to bind controlling devices with devices being controlled, such as switches and lights. Particularly, end device bind supports a simplified binding method where user input is utilized to identify controlling/controlled device pairs. Bind and unbind services provide the ability for creation and deletion of binding table entry that map control messages to their intended destination.

Network management services provide the ability to retrieve management information from the devices, including network discovery results, routing table contents, link quality to neighbor nodes, and binding table contents. It also provides the ability to control the network association by disassociating devices from the PAN. Network management services are designed majorly for user or commissioning tools to manage the network. APIs from each of these three categories are discussed in the following sub-sections.

3.2.2 ZDO Device Network Startup

By default `ZDApp_Init()` [in `ZDApp.c`] starts the device's startup in a Zigbee network, but an application can override this default behavior. The compilation flag `HOLD_AUTO_START` is not longer used, the compile flag `NV_RESTORE` is by default enabled. It is suggested that the user do not use the ZDO interface to commission the device to the network, but instead use the BDB API.

3.2.2.1 ZDOInitDeviceEx()

Start the device in the network. This function will read `ZCD_NV_STARTUP_OPTION` (NV item) to determine whether or not to restore the network state of the device.

If the application would like to force a "new" join, the application should set the `ZCD_STARTOPT_DEFAULT_NETWORK_STATE` bit in the `ZCD_NV_STARTUP_OPTION` NV item before calling this function. "New" join means to not restore the network state of the device. Use

zgWriteStartupOptions() to set these options

```
[zgWriteStartupOptions(ZG_STARTUP_SET, ZCD_STARTOPT_DEFAULT_NETWORK_STATE);].
```

For Router devices the parameter *mode* indicates if the Router will create a distributed network or will perform joining. The formation procedure can only be performed if the previous network configuration is ignored by setting the device to “new”.

Prototype

```
uint8 ZDOInitDeviceEx( uint16 startDelay, uint8 mode);
```

Parameter Details

startDelay – the delay to start device (in milliseconds). There is a jitter added to this delay:

```
((NWK_START_DELAY + startDelay)
+ (osal_rand() & EXTENDED_JOINING_RANDOM_MASK))
```

mode – For Routers, if TRUE and the device is configured as “new”, then it will form a distributed network, otherwise attempt joining. For non Router devices this parameter does not have effect.

Return

This function will return one of the following:

Name	Description
ZDO_INITDEV_RESTORED_NETWORK_STATE	The device's network state was restored.
ZDO_INITDEV_NEW_NETWORK_STATE	The network state was initialized. This could mean that ZCD_NV_STARTUP_OPTION said to not restore, or it could mean that there was no network state to restore.
ZDO_INITDEV_LEAVE_NOT_STARTED	Before the reset, a network leave was issued with the rejoin option set to TRUE. So, the device was not started in the network (one time only). The next time this function is called it will start.

3.2.2.2 ZDOInitDevice()

Macro function for ZDOInitDeviceEx() set to behave as the legacy function.

3.2.3 ZDO Message Callbacks

An application can receive any over the air message (request or response) by registering for it with ZDO_RegisterForZDOMsg().

3.2.3.1 ZDO_RegisterForZDOMsg()

Call this function to request an over-the-air message. A copy of the message will be sent to a task in an OSAL message. The task receiving the message can either parse the message themselves or call a ZDO Parser function to parse the message. Only response messages have a ZDO Parser function.

After registering for a message, and the message is received (Over-The-Air), the message is sent to the application/task as a ZDO_CB_MSG (OSAL Msg). The body of the message (zdoIncomingMsg_t – defined in ZDProfile.h) contains the Over-The-Air message.

Prototype

```
ZStatus_t ZDO_RegisterForZDOMsg( uint8 taskID, uint16 clusterID );
```

Parameter Details

taskID – the application’s task ID. This will be used to send the OSAL message.

clusterID – the over the air message’s clusterID that you would like to receive (example: NWK_addr_rsp). These are defined in ZDProfile.h.

Return

ZStatus_t –status values defined in ZStatus_t in ZComDef.h.

3.2.3.2 ZDO_RemoveRegisteredCB()

Call this function to remove a request for an over-the-air message.

Prototype

```
ZStatus_t ZDO_RemoveRegisteredCB( uint8 taskID, uint16 clusterID );
```

Parameter Details

taskID – the application’s task ID. Must be the same task ID used when ZDO_RegisterForZDOMsg() was called.

clusterID – the over the air message’s clusterID. Must be the same task ID used when ZDO_RegisterForZDOMsg() was called.

Return

ZStatus_t –status values defined in ZStatus_t in ZComDef.h.

3.2.4 ZDO Discovery API

The ZDO Discovery API builds and sends ZDO device and service discovery requests and responses. All API functions and their corresponding ZDP commands are listed in the following table. User can use the command name as the keyword to search in the latest ZigBee Specification for further reference. Each of these functions will be discussed in detail in the following sub-clauses.

ZDO API Function	ZDP Discovery Command
ZDP_NwkAddrReq()	NWK_addr_req
ZDP_NWKAddrRsp()	NWK_addr_rsp

ZDP_IEEEAddrReq()	IEEE_addr_req
ZDP_IEEEAddrRsp()	IEEE_addr_rsp
ZDP_NodeDescReq()	Node_Desc_req
ZDP_NodeDescRsp()	Node_Desc_rsp
ZDP_PowerDescReq()	Power_Desc_req
ZDP_PowerDescRsp()	Power_Desc_rsp
ZDP_SimpleDescReq()	Simple_Desc_req
ZDP_SimpleDescRsp()	Simple_Desc_rsp
ZDP_ComplexDescReq()	Complex_Desc_req
ZDP_ActiveEPIFReq()	Active_EP_req
ZDP_ActiveEPIFRsp()	Active_EP_rsp
ZDP_MatchDescReq()	Match_Desc_req
ZDP_MatchDescRsp()	Match_Desc_rsp
ZDP_UserDescSet()	User_Desc_set
ZDP_UserDescConf()	User_Desc_conf
ZDP_UserDescReq()	User_Desc_req
ZDP_UserDescRsp()	User_Desc_rsp
ZDP_DeviceAnnce()	Device_annce
ZDP_ServerDiscReq()	System_Server_Discovery_req
ZDP_ServerDiscRsp()	System_Server_Discovery_rsp

3.2.4.1 ZDP_NwkAddrReq()

Calling this function will generate a message to ask for the 16 bit address of the Remote Device based on its known IEEE address. This message is sent as a broadcast message to all devices in the network.

Prototype

```
afStatus_t ZDP_NwkAddrReq( byte *IEEEAddress, byte ReqType, byte StartIndex, byte SecuritySuite );
```

Parameter Details

IEEEAddress – The IEEE address of the device under request.

`ReqType` - type of response wanted. Its possible values are listed in the following table:

Name	Meaning
ZDP_NWKADDR_REQTYPE_SINGLE	Return only the device's short and extended address
ZDP_NWKADDR_REQTYPE_EXTENDED	Return the device's short and extended address and the short address of all associated devices.

`StartIndex` – the responding device could have more response items than can fit in a response message, the requester can specify a starting index into the possible response items. Index starts from zero.

`SecuritySuite` - Type of security wanted on the message.

Return

`afStatus_t` – This function uses AF to send the message, so the status values are AF status values defined in `ZStatus_t` in `ZComDef.h`.

3.2.4.2 ZDP_NWKAddrRsp()

This is actually a macro that calls `ZDP_AddrRsp()`. This call will build and send a Network Address Response.

Prototype

```
afStatus_t ZDP_NWKAddrRsp( byte TranSeq, zAddrType_t *dstAddr,
    byte Status, byte *IEEEAddrRemoteDev,
    byte ReqType, uint16 nwkAddr,
    byte NumAssocDev, byte StartIndex,
    uint16 *NWKAddrAssocDevList,
    byte SecuritySuite );
```

Parameter Details

`TranSeq` – The transaction sequence number,

`DstAddr` - The destination address.

`Status` – The following values:

Status	
Meaning	Value
ZDP_SUCCESS	0
ZDP_INVALID_REQTYPE	1
ZDP_DEVICE_NOT_FOUND	2
Reserved	0x03-0xff

`IEEEAddrRemoteDev` – the 64 bit address for the remote device.

`ReqType` – the request type of the request.

`nwkAddr` – the 16 bit address for the remote device.

`NumAssocDev` – Count of the number of associated devices to the Remote Device and the number of 16 bit short addresses to follow. `NumAssocDev` shall be 0 if there are no associated devices to Remote Device and the `StartIndex` and `NWKAddrAssocDevList` shall be null in this case.

`StartIndex` – the responding device could have more response items than can fit in a response message, the requester can specify a starting index into the possible response items. This field is the starting index for this response message.

`NWKAddrAssocDevList` – A list of 16 bit addresses, one corresponding to each associated device to the Remote Device. The count of the 16 bit addresses in `NWKAddrAssocDevList` is supplied in `NumAssocDev`.

`SecuritySuite` - Type of security wanted on the message.

Return

`afStatus_t` – This function uses AF to send the message, so the status values are AF status values defined in `ZStatus_t` in `ZComDef.h`.

3.2.4.3 ZDP_IEEEAddrReq()

Calling this function will generate a message to ask for the 64 bit address of the Remote Device based on its known 16 bit network address.

Prototype

`afStatus_t ZDP_IEEEAddrReq(uint16 shortAddr, byte ReqType, byte StartIndex, byte SecuritySuite);`

Parameter Details

`shortAddr` – known 16 bit network address.

`ReqType` - type of response wanted.

ReqType	
Name	Meaning
<code>ZDP_IEEEADDR_REQTYPE_SINGLE</code>	Return only the device's short and extended address
<code>ZDP_IEEEADDR_REQTYPE_EXTENDED</code>	Return the device's short and extended address and the short address of all associated devices.

`StartIndex` – the responding device could have more response items than can fit in a response message, the requester can specify a starting index into the possible response items.

`SecuritySuite` - Type of security wanted on the message.

Return

`afStatus_t` – This function uses AF to send the message, so the status values are AF status values defined in `ZStatus_t` in `ZComDef.h`.

3.2.4.4 ZDP_IEEEAddrRsp()

This is actually a macro that calls `ZDP_AddrRsp()`. This call will build and send an IEEE Address Response.

Prototype

```
afStatus_t ZDP_IEEEAddrRsp( byte TranSeq, zAddrType_t *dstAddr,
                             byte Status, byte *IEEEAddrRemoteDev,
                             byte ReqType, uint16 nwkAddr,
                             byte NumAssocDev, byte StartIndex,
                             uint16 *NWKAddrAssocDevList,
                             byte SecuritySuite );
```

Parameter Details

`TranSeq` – The transaction sequence number.

`DstAddr` – The destination address.

`Status` – The following values:

Status	
Meaning	Value
ZDP_SUCCESS	0
ZDP_INVALID_REQTYPE	1
ZDP_DEVICE_NOT_FOUND	2
Reserved	0x03-0xff

`IEEEAddrRemoteDev` – the 64 bit address for the remote device.

`ReqType` – the request type of the request.

`nwkAddr` – the 16 bit address for the remote device.

`NumAssocDev` – Count of the number of associated devices to the Remote Device and the number of 16 bit short addresses to follow. `NumAssocDev` shall be 0 if there are no associated devices to Remote Device and the `StartIndex` and `NWKAddrAssocDevList` shall be null in this case.

`StartIndex` – Starting index into the list of associated devices for this report.

`NWKAddrAssocDevList` – A list of 16 bit addresses, one corresponding to each associated device to the Remote Device. The count of the 16 bit addresses in `NWKAddrAssocDevList` is supplied in `NumAssocDev`.

`SecuritySuite` – Type of security wanted on the message.

Return

`afStatus_t` – This function uses AF to send the message, so the status values are AF status values defined in `ZStatus_t` in `ZComDef.h`.

3.2.4.5 ZDP_NodeDescReq()

This is actually a macro that calls `ZDP_NWKAddrOfInterestReq()`. This call will build and send a Node Descriptor Request to the Remote Device specified in the destination address field.

Prototype

```
afStatus_t ZDP_NodeDescReq( zAddrType_t *dstAddr, uint16 NWKAddrOfInterest, byte SecuritySuite );
```

Parameter Details

`DstAddr` - The destination address.

`NWKAddrOfInterest` - The 16 bit short address to search.

`SecuritySuite` - Type of security wanted on the message.

Return

`afStatus_t` – This function uses AF to send the message, so the status values are AF status values defined in `ZStatus_t` in `ZComDef.h`.

3.2.4.6 ZDP_NodeDescMsg()

Call this function to respond to the Node Descriptor Request.

Prototype

```
afStatus_t ZDP_NodeDescMsg( byte TransSeq, zAddrType_t *dstAddr, byte Status,
                             uint16 nwkAddr, NodeDescriptorFormat_t *pNodeDesc,
                             byte SecuritySuite );
```

Parameter Details

`TranSeq` – The transaction sequence number.

`DstAddr` - The destination address.

`Status` – The following results:

Status	
Meaning	Value
SUCCESS	0
DEVICE_NOT_FOUND	1

`nwkAddr` - the device's 16 bit address .
`pNodeDesc` - pointer to the node descriptor (defined in AF.h).
`SecuritySuite` - Type of security wanted on the message.

Return

`afStatus_t` - This function uses AF to send the message, so the status values are AF status values defined in `ZStatus_t` in `ZComDef.h`.

3.2.4.7 ZDP_PowerDescReq()

This is actually a macro that calls `ZDP_NWKAddrOfInterestReq()`. This call will build and send a Power Descriptor Request. Use this macro to ask for the Power Descriptor of the Remote Device.

Prototype

```
afStatus_t ZDP_PowerDescReq( zAddrType_t *dstAddr, int16 NWKAddrOfInterest, byte SecuritySuite );
```

Parameter Details

`DstAddr` - The destination address.
`NWKAddrOfInterest` - The 16 bit short address to search.
`SecuritySuite` - Type of security wanted on the message.

Return

`afStatus_t` - This function uses AF to send the message, so the status values are AF status values defined in `ZStatus_t` in `ZComDef.h`.

3.2.4.8 ZDP_PowerDescMsg()

Call this function to respond to the Power Descriptor Request.

Prototype

```
afStatus_t ZDP_PowerDescMsg( byte TranSeq, zAddrType_t *dstAddr, byte Status, int16 nwkAddr,  
NodePowerDescriptorFormat_t *pPowerDesc,  
byte SecuritySuite );
```

Parameter Details

`TranSeq` - The transaction sequence number.
`DstAddr` - The destination address.
`Status` - The following results:

Status	Value
SUCCESS	0

DEVICE_NOT_FOUND	1
------------------	---

nwkAddr - the device's 16 bit address .

pPowerDesc - pointer to the power descriptor (defined in AF.h).

SecuritySuite - Type of security wanted on the message.

Return

afStatus_t - This function uses AF to send the message, so the status values are AF status values defined in ZStatus_t in ZComDef.h.

3.2.4.9 ZDP_SimpleDescReq()

This call will build and send a Simple Descriptor Request.

Prototype

```
afStatus_t ZDP_SimpleDescReq( zAddrType_t *dstAddr, uint16 nwkAddr, byte epIntf, byte SecuritySuite );
```

Parameter Details

DstAddr - The destination address.

nwkAddr - Known 16 bit network address.

epIntf - wanted application's endpoint/interface.

SecuritySuite - Type of security wanted on the message.

Return

afStatus_t - This function uses AF to send the message, so the status values are described in ZStatus_t in ZComDef.h.

3.2.4.10 ZDP_SimpleDescRsp()

Call this function to respond to the Simple Descriptor Request.

Prototype

```
afStatus_t ZDP_SimpleDescRsp( byte TranSeq, zAddrType_t *dstAddr,
byte Status, SimpleDescriptionFormat_t *pSimpleDesc,
byte SecuritySuite );
```

Parameter Details

TranSeq - The transaction sequence number.

DstAddr - The destination address.

Status - The following results:

Status	Value
SUCCESS	0
INVALID_EP	1
NOT_ACTIVE	2
DEVICE_NOT_FOUND	3

`pSimpleDesc` - pointer to the simple descriptor (defined in AF.h).

`SecuritySuite` - Type of security wanted on the message.

Return

`afStatus_t` - This function uses AF to send the message, so the status values are AF status values defined in `ZStatus_t` in `ZComDef.h`.

3.2.4.11 ZDP_ComplexDescReq()

This call will build and send a Complex Descriptor Request. It is a macro that calls `ZDP_NWKAddrOfInterestReq()`.

Prototype

```
afStatus_t ZDP_ComplexDescReq( zAddrType_t *dstAddr, uint16 nwkAddr, byte SecuritySuite );
```

Parameter Details

`DstAddr` - The destination address.

`nwkAddr` - Known 16 bit network address.

`SecuritySuite` - Type of security wanted on the message.

Return

`afStatus_t` - This function uses AF to send the message, so the status values are described in `ZStatus_t` in `ZComDef.h`.

3.2.4.12 ZDP_ActiveEPIFReq ()

This is actually a macro that calls `ZDP_NWKAddrOfInterestReq()`. This call will build and send an Active Endpoint/Interface Request. Use this macro ask for all the active endpoint/interfaces on the Remote Device.

Prototype

```
afStatus_t ZDP_ActiveEPIFReq( zAddrType_t *dstAddr, uint16 NWKAddrOfInterest, byte SecuritySuite );
```

Parameter Details

`DstAddr` - The destination address.

`NWKAddrOfInterest` - The 16 bit short address to search.

`SecuritySuite` - Type of security wanted on the message.

Return

`afStatus_t` - This function uses AF to send the message, so the status values are AF status values defined in `ZStatus_t` in `ZComDef.h`.

3.2.4.13 ZDP_ActiveEPIFRsp()

This is a macro that calls `ZDP_EPIFRsp()`. Call this function to respond to the Active Endpoint/Interface Request.

Prototype

```
afStatus_t ZDP_ActiveEPIFRsp( byte TranSeq, zAddrType_t *dstAddr,
byte Status, uint16 nwkAddr, byte Count, byte *pEPIntfList, byte SecuritySuite );
```

Parameter Details

`TranSeq` - The transaction sequence number.

`DstAddr` - The destination address.

`Status` - The following results:

Status	
Meaning	Value
SUCCESS	0
DEVICE_NOT_FOUND	1

`nwkAddr` - device's 16 bit network address.

`Count` - number to active endpoint/interfaces in `pEPIntfList`

`pEPIntfList` - array of active endpoint/interfaces on this device.

`SecuritySuite` - Type of security wanted on the message.

Return

`afStatus_t` - This function uses AF to send the message, so the status values are AF status values defined in `ZStatus_t` in `ZComDef.h`.

3.2.4.14 ZDP_MatchDescReq()

This call will build and send an Match Description Request. Use this function to search for devices/applications that match something in the input/output cluster list of an application.

Prototype

```
afStatus_t ZDP_MatchDescReq( zAddrType_t *dstAddr, uint16 nwkAddr,
uint16 ProfileID,
byte NumInClusters, byte *InClusterList,
byte NumOutClusters, byte *OutClusterList,
byte SecuritySuite );
```

Parameter Details

`DstAddr` - The destination address.

`nwkAddr` – known 16 bit network address.

`ProfileID` – application’s profile ID as a reference for the cluster IDs.

`NumInClusters` – number of cluster IDs in the input cluster list.

`InClusterList` – array of input cluster IDs (each 1 byte).

`NumOutClusters` – number of cluster IDs in the output cluster list.

`OutClusterList` – array of output cluster IDs (each 1 byte).

`SecuritySuite` - Type of security wanted on the message.

Return

`afStatus_t` – This function uses AF to send the message, so the status values are AF status values defined in `ZStatus_t` in `ZComDef.h`.

3.2.4.15 ZDP_MatchDescRsp()

This is a macro that calls `ZDP_EPIFRsp()`. Call this function to respond to the Match Description Request.

Prototype

```
afStatus_t ZDP_MatchDescRsp( byte TranSeq, zAddrType_t *dstAddr, byte Status, uint16 nwkAddr, byte Count,
byte *pEPIntfList, byte SecuritySuite );
```

Parameter Details

`TranSeq` - The transaction sequence number.

`DstAddr` - The destination address.

`Status` – The following results:

Status	
Meaning	Value
SUCCESS	0
DEVICE_NOT_FOUND	1

`nwkAddr` - device’s 16 bit network address.

`Count` - number to active endpoint/interfaces in `pEPIntfList`

`pEPIntfList` - array of active endpoint/interfaces on this device.

`SecuritySuite` - Type of security wanted on the message.

Return

`afStatus_t` - This function uses AF to send the message, so the status values are AF status values defined in `ZStatus_t` in `ZComDef.h`.

3.2.4.16 ZDP_UserDescSet()

This function builds and sends a `User_Desc_set` message to set the user descriptor for the remote device. The request is unicast to the remote device itself or other devices that have the discovery information of the remote device. Notice that the remote device shall have `NV_RESTORE` defined to enable this functionality.

Prototype

```
afStatus_t ZDP_UserDescSet( zAddrType_t *dstAddr,
                           uint16 nwkAddr,
                           UserDescriptorFormat_t *UserDescriptor,
                           byte SecurityEnable );
```

Parameter Details

`dstAddr` - destination address for the request

`nwkAddr` - 16 bits NWK address for the remote device of interest.

`UserDescriptor` - The user descriptor to be configured. It contains an ASCII character string with length less than or equal to 16 characters. It will be padded with space characters (0x20) to make a total length of 16 characters.

`SecurityEnable` - Type of security wanted on the message.

Return

`afStatus_t` - This function uses AF to send the message, so the status values are AF status values defined in `ZStatus_t` in `ZComDef.h`.

3.2.4.17 ZDP_UserDescConf()

This is a macro that calls `ZDP_SendData ()` directly. Call this function to respond to the `User_Desc_Conf`

Prototype

```
afStatus_t ZDP_UserDescConf( byte TranSeq, zAddrType_t *dstAddr,
                             byte Status, byte SecurityEnable );
```

Parameter Details

`TranSeq` - The transaction sequence number.

`DstAddr` - The destination address.

Status – The following results:

Status	
Meaning	Value
SUCCESS	0x00
INV_REQUESTTYPE	0x80
DEVICE_NOT_FOUND	0x81
NOT_SUPPORTED	0x84

SecurityEnable - Type of security wanted on the message.

Return

afStatus_t – This function uses AF to send the message, so the status values are AF status values defined in ZStatus_t in ZComDef.h.

3.2.4.18 ZDP_UserDescReq()

This call will build and send a User_Desc_Req. It is a macro that calls ZDP_NWKAddrOfInterestReq().

Prototype

```
afStatus_t ZDP_UserDescReq( zAddrType_t *dstAddr, uint16 nwkAddr,
                           byte SecurityEnable );
```

Parameter Details

DstAddr - The destination address.

nwkAddr – Known 16 bit network address.

SecurityEnable - Type of security wanted on the message.

Return

afStatus_t – This function uses AF to send the message, so the status values are described in ZStatus_t in ZComDef.h.

3.2.4.19 ZDP_UserDescRsp()

This call will build and send a User_Desc_Rsp.

Prototype

```
ZStatus_t ZDP_UserDescRsp( byte TransSeq, zAddrType_t *dstAddr,
                           uint16 nwkAddrOfInterest, UserDescriptorFormat_t *userDesc,
                           byte SecurityEnable );
```

Parameter Details

TranSeq – The transaction sequence number.

DstAddr – The destination address.

nwkAddrOfInterest – Known 16 bit network address.

userDesc – User Descriptor of the local device.

SecurityEnable – Type of security wanted on the message.

Return

afStatus_t – This function uses AF to send the message, so the status values are described in ZStatus_t in ZComDef.h.

3.2.4.20 ZDP_DeviceAnnce()

This function builds and sends a Device_annce command for ZigBee end device to notify other ZigBee devices on the network that the end device has joined or rejoined the network. The command contains the device's new 16-bit NWK address and its 64-bit IEEE address, as well as the capabilities of the ZigBee device. It is sent out as broadcast message.

On receipt of the Device_annce, all receivers shall check all internal references to the IEEE address supplied in the announce, and substitute the corresponding NWK address with the new one. No response will be sent back for Device_annce.

Prototype

```
afStatus_t ZDP_DeviceAnnce( uint16 nwkAddr, byte *IEEEAddr, byte capabilities, byte SecurityEnable );
```

Parameter Details

nwkAddr – 16 bits NWK address for the local device.

IEEEAddr – pointer to the 64 bits IEEE address for the local device.

Capabilities – Capability of the local device.

SecurityEnable – Type of security wanted on the message.

Return

afStatus_t – This function uses AF to send the message, so the status values are AF status values defined in ZStatus_t in ZComDef.h.

3.2.4.21 ZDP_ServerDiscReq()

The function builds and sends a System_Server_Discovery_req request message which contains a 16 bit server mask. The purpose of this request is to discover the locations of a particular system server or servers as indicated by the server mask. The message is broadcast to all device with RxOnWhenIdle. Remote devices will send responses back only if a match bit is found when comparing the received server mask with the mask stored in the local node descriptor, using unicast transmission.

Prototype

```
afStatus_t ZDP_ServerDiscReq( uint16 serverMask, byte SecurityEnable );
```

Parameter Details

serverMask – 16-bit bit mask of system servers being sought

SecurityEnable – Type of security wanted on the message.

Return

afStatus_t – This function uses AF to send the message, so the status values are AF status values defined in ZStatus_t in ZComDef.h.

3.2.4.22 ZDP_ServerDiscRsp()

The function builds and sends back System_Server_Discovery_rsp response. It will be called when a System_Server_Discovery_req is received and a match is found for the server bit mask.

Prototype

```
ZStatus_t ZDP_ServerDiscRsp( byte transID, zAddrType_t *dstAddr, byte status, uint16 aoI, uint16 serverMask,  
                             byte SecurityEnable );
```

Parameter Details

transID – ZDO transaction sequence number

dstAddr – destination of the response

status – The status is always ZSUCCESS.

aoI – address of interest. Currently it is not used.

serverMask – 16 bit bit-mask which indicated matched system server.

securityEnable - Type of security wanted on the message.

Return

afStatus_t – This function uses AF to send the message, so the status values are AF status values defined in ZStatus_t in ZComDef.h.

3.2.5 ZDO Binding API

The ZDO Binding API builds and sends ZDO binding requests and responses. All binding information (tables) are kept in the Zigbee Coordinator. So, only the Zigbee Coordinator can receive bind requests. The table below lists the Binding API supported by the stack and their corresponding command name in ZigBee Specification. . User can use the command name as the keyword to search in the latest ZigBee Specification for further reference. Each of these primitives will be discussed in the following sub-clauses.

ZDO Binding API	ZDP Binding Service Command
ZDP_EndDeviceBindReq()	End_Device_Bind_req
ZDP_EndDeviceBindRsp()	End_Device_Bind_rsp
ZDP_BindReq()	Bind_req
ZDP_BindRsp()	Bind_rsp
ZDP_UnbindReq()	Unbind_req
ZDP_UnbindRsp()	Unbind_rsp

3.2.5.1 ZDP_EndDeviceBindReq()

This call will build and send an End Device Bind Request (“Manual Binding”). Send this message to attempt a manually generated bind for this device. After manual binding you can send indirect (no address) message to the coordinator and the coordinator will send the message to the device that this message is bound to, or you will receive messages from your new bound device.

Prototype

```
afStatus_t ZDP_EndDeviceBindReq( zAddrType_t *dstAddr,
                                uint16 LocalCoordinator,
                                byte ep,
                                uint16 ProfileID,
                                byte NumInClusters, byte *InClusterList,
                                byte NumOutClusters, byte *OutClusterList,
                                byte SecuritySuite );
```

Parameter Details

DstAddr - The destination address.

LocalCoordinator – Known 16 bit network address of the device’s parent coordinator.

ep – the application’s endpoint/interface.

ProfileID – the application’s profile ID as reference for the Cluster IDs.

NumInClusters – number of cluster IDs in the input cluster list.

InClusterList – array of input cluster IDs (each 1 byte).

NumOutClusters – number of cluster IDs in the output cluster list.

OutClusterList – array of output cluster IDs (each 1 byte).

`SecuritySuite` - Type of security wanted on the message.

Return

`afStatus_t` – This function uses AF to send the message, so the status values are AF status values defined in `ZStatus_t` in `ZComDef.h`.

3.2.5.2 ZDP_EndDeviceBindRsp()

This is a macro that calls `ZDP_SendData()` directly. Call this function to respond to the End Device Bind Request.

Prototype

```
afStatus_t ZDP_EndDeviceBindRsp( byte TranSeq, zAddrType_t *dstAddr, byte Status, byte SecurityEnable );
```

Parameter Details

`TranSeq` – The transaction sequence number.

`DstAddr` - The destination address.

`Status` – The following results:

Status	
Meaning	Value
SUCCESS	0
NOT_SUPPORTED	1
TIMEOUT	2
NO_MATCH	3

`SecurityEnable` - Type of security wanted on the message.

Return

`afStatus_t` – This function uses AF to send the message, so the status values are AF status values defined in `ZStatus_t` in `ZComDef.h`.

3.2.5.3 ZDP_BindReq()

This is actually a macro that calls `ZDP_BindUnbindReq()`. This call will build and send a Bind Request. Use this function to request the Zigbee Coordinator to bind application based on clusterID.

Prototype

```
afStatus_t ZDP_BindReq( zAddrType_t *dstAddr, byte *SourceAddr,
                        byte SrcEP, byte ClusterID, byte *DestinationAddr, byte DstEP, byte SecuritySuite );
```

Parameter Details

`DstAddr` - The destination address.

`SourceAddr` - 64 bit IEEE address of the device generating the messages.

`SrcEP` - endpoint of the application generating the messages.

`ClusterID` - cluster ID of the messages to bind.

`DestinationAddr` - 64 bit IEEE address of the device receiving the messages.

`DstEP` - endpoint of the application receiving the messages.

`SecuritySuite` - Type of security wanted on the message.

Return

`afStatus_t` - This function uses AF to send the message, so the status values are AF status values defined in `ZStatus_t` in `ZComDef.h`.

3.2.5.4 ZDP_BindRsp()

This is a macro that calls `ZDP_SendData()` directly. Call this function to respond to the Bind Request.

Prototype

`afStatus_t ZDP_BindRsp(byte TranSeq, zAddrType_t *dstAddr, byte Status, byte SecurityEnable);`

Parameter Details

`TranSeq` - The transaction sequence number.

`DstAddr` - The destination address.

`Status` - The following results:

Status	
Meaning	Value
SUCCESS	0
NOT_SUPPORTED	1
TABLE_FULL	2

`SecurityEnable` - Type of security wanted on the message.

Return

`afStatus_t` - This function uses AF to send the message, so the status values are AF status values defined in `ZStatus_t` in `ZComDef.h`.

3.2.5.5 ZDP_UnbindReq()

This is actually a macro that calls `ZDP_BindUnbindReq()`. This call will build and send a Unbind Request. Use this function to request the Zigbee Coordinator's removal of a binding.

Prototype

```
afStatus_t ZDP_UnbindReq( zAddrType_t *dstAddr, byte *SourceAddr, byte SrcEP, byte ClusterID,
                          byte *DestinationAddr, byte DstEP, byte SecuritySuite );
```

Parameter Details

`DstAddr` - The destination address.

`SourceAddr` - 64 bit IEEE address of the device generating the messages.

`SrcEP` - endpoint of the application generating the messages.

`ClusterID` - cluster ID of the messages to bind.

`DestinationAddr` - 64 bit IEEE address of the device receiving the messages.

`DstEP` - endpoint of the application receiving the messages.

`SecuritySuite` - Type of security wanted on the message.

Return

`afStatus_t` - This function uses AF to send the message, so the status values are AF status values defined in `ZStatus_t` in `ZComDef.h`.

3.2.5.6 ZDP_UnbindRsp()

This is a macro that calls `ZDP_SendData ()` directly. Call this function to respond to the Unbind Request.

Prototype

```
afStatus_t ZDP_UnbindRsp( byte TranSeq, zAddrType_t *dstAddr, byte Status, byte SecurityEnable );
```

Parameter Details

`TranSeq` - The transaction sequence number.

`DstAddr` - The destination address.

`Status` - The following results:

Status	
Meaning	Value
SUCCESS	0
NOT_SUPPORTED	1
NO_ENTRY	2

`SecurityEnable` - Type of security wanted on the message.

Return

`afStatus_t` - This function uses AF to send the message, so the status values are AF status values defined in `ZStatus_t` in `ZComDef.h`.

3.2.6 ZDO Management API

The ZDO Management API builds and sends ZDO Management requests and responses. These messages are used to get device status and update tables. The table below lists the Management API supported by the stack and their corresponding command name in ZigBee Specification. . User can use the command name as the keyword to search in the latest ZigBee Specification for further reference. Each of these primitives will be discussed in the following sub-clauses.

ZDP Management API	ZDP Network Management Service Command
ZDP_MgmtNwkDiscReq()	Mgmt_NWK_Disc_req
ZDP_MgmtNwkDiscRsp()	Mgmt_NWK_Disc_rsp
ZDP_MgmtLqiReq()	Mgmt_Lqi_req
ZDP_MgmtLqiRsp()	Mgmt_Lqi_rsp
ZDP_MgmtRtgReq()	Mgmt_Lqi_req
ZDP_MgmtRtgRsp()	Mgmt_Rtg_rsp
ZDP_MgmtBindReq()	Mgmt_Bind_req
ZDP_MgmtBindRsp()	Mgmt_Bind_rsp
ZDP_MgmtLeaveReq()	Mgmt_Leave_req
ZDP_MgmtLeaveRsp()	Mgmt_Leave_rsp
ZDP_MgmtDirectJoinReq()	Mgmt_Direct_Join_req
ZDP_MgmtDirectJoinRsp()	Mgmt_Direct_Join_rsp
ZDP_MgmtPermitJoinReq()	Mgmt_Permit_Join_req
ZDP_MgmtPermitJoinRsp()	Mgmt_Permit_Join_rsp

3.2.6.1 ZDP_MgmtNwkDiscReq()

If the device supports this command (optional), calling this function will generate the request for the destination device to perform a network scan. The calling application can only call this function if the ZDO_MGMT_NWKDISC_REQUEST compile flag is set either in ZDConfig.h or as a normal compile flag.

Prototype

```
afStatus_t ZDP_MgmtNwkDiscReq( zAddrType_t *dstAddr,
                               uint32 ScanChannels, byte StartIndex, byte SecurityEnable );
```

Parameter Details

DstAddr - The destination address.

`ScanChannels` – Bit mask containing the channels to scan for this request. The channel definitions are contained in `NLMEDE.h` (ex. `DEFAULT_CHANLIST`).

`StartIndex` – the responding device could have more response items than can fit in a response message, the requester can specify a starting index into the possible response items.

`SecurityEnable` – true if security is enabled.

Return

`afStatus_t` – This function uses AF to send the message, so the status values are AF status values defined in `ZStatus_t` in `ZComDef.h`.

3.2.6.2 ZDP_MgmtNwkDiscRsp()

If the device supports this command (optional), calling this function will generate the response. The ZDO will generate this message automatically when a “Management Network Discovery Request” message is received if the `ZDO_MGMT_NWKDISC_RESPONSE` compile flag is set either in `ZDConfig.h` or as a normal compile flag.

Prototype

```
afStatus_t ZDP_MgmtNwkDiscRsp( byte TranSeq, zAddrType_t *dstAddr,
                               byte Status,
                               byte NetworkCount, byte StartIndex, byte NetworkCountList,
                               networkDesc_t *NetworkList, byte SecurityEnable );
```

Parameter Details

`TranSeq` – The transaction sequence number.

`DstAddr` – The destination address.

`Status` – defined in `ZComDef.h` as `ZStatus_t`.

`NetworkCount` – The number of possible items in this message.

`StartIndex` – the responding device could have more response items than can fit in a response message, the requester can specify a starting index into the possible response items. This field is the starting index for this response message.

`NetworkCountList` – The number of response items in this message.

`NetworkList` – the list of Network Discovery records. You can look up `networkDesc_t` in `NLMEDE.h` for information on this structure.

`SecurityEnable` – true if security is enabled.

Return

`afStatus_t` – This function uses AF to send the message, so the status values are AF status values defined in `ZStatus_t` in `ZComDef.h`.

3.2.6.3 ZDP_MgmtLqiReq()

If the device supports this command (optional), calling this function will generate the request for the destination device to return its neighbor list. The calling application can only call this function if the ZDO_MGMT_LQI_REQUEST compile flag is set either in ZDConfig.h or as a normal compile flag.

Prototype

```
afStatus_t ZDP_MgmtLqiReq ( zAddrType_t *dstAddr, byte StartIndex, byte SecurityEnable );
```

Parameter Details

`DstAddr` - The destination address.

`StartIndex` – the responding device could have more response items than can fit in a response message, the requester can specify a starting index into the possible response items.

`SecurityEnable` – true if security is enabled.

Return

`afStatus_t` – This function uses AF to send the message, so the status values are AF status values defined in `ZStatus_t` in `ZComDef.h`.

3.2.6.4 ZDP_MgmtLqiRsp()

If the device supports this command (optional), calling this function will generate the response. The ZDO will generate this message automatically when a “Management LQI Request” message is received if the ZDO_MGMT_LQI_RESPONSE compile flag is set either in ZDConfig.h or as a normal compile flag.

Prototype

```
ZStatus_t ZDP_MgmtLqiRsp( byte TranSeq, zAddrType_t *dstAddr,  
                          byte Status, byte NeighborLqiEntries,  
                          byte StartIndex, byte NeighborLqiCount,  
                          neighborLqiItem_t *NeighborLqiList,  
                          byte SecurityEnable );
```

Parameter Details

`TranSeq` – The transaction sequence number.

`DstAddr` - The destination address.

`Status` – defined in `ZComDef.h` as `ZStatus_t`.

`NeighborLqiEntries` – The number of possible items in this message.

`StartIndex` – the responding device could have more response items than can fit in a response message, the requester can specify a starting index into the possible response items. This field is the starting index for this response message.

`NeighborLqiCount` – The number of response items in this message.

`NeighborLqiList` – the list of neighbor records. You can look up `neighborLqiItem_t` in `ZDProfile.h` for information on this structure.

`SecurityEnable` – true if security is enabled.

Return

`ZStatus_t` – status values defined in `ZStatus_t` in `ZComDef.h`.

3.2.6.5 ZDP_MgmtRtgReq ()

If the device supports this command (optional), calling this function will generate the request for the destination device to return its Routing Table. The calling application can only call this function if the `ZDO_MGMT_RTG_REQUEST` compile flag is set either in `ZDConfig.h` or as a normal compile flag.

Prototype

```
afStatus_t ZDP_MgmtRtgReq( zAddrType_t *dstAddr, byte StartIndex, byte SecurityEnable );
```

Parameter Details

`DstAddr` – The destination address.

`StartIndex` – the responding device could have more response items than can fit in a response message, the requester can specify a starting index into the possible response items.

`SecurityEnable` – true if security is enabled.

Return

`afStatus_t` – This function uses AF to send the message, so the status values are AF status values defined in `ZStatus_t` in `ZComDef.h`.

3.2.6.6 ZDP_MgmtRtgRsp()

If the device supports this command (optional), calling this function will generate the response. The ZDO will generate this message automatically when a “Management Routing Request” message is received if the `ZDO_MGMT_RTG_RESPONSE` compile flag is set either in `ZDConfig.h` or as a normal compile flag.

Prototype

```
ZStatus_t ZDP_MgmtRtgRsp( byte TranSeq, zAddrType_t *dstAddr,
                          byte Status, byte RoutingTableEntries,
                          byte StartIndex, byte RoutingListCount,
                          rtgItem_t *RoutingTableList, byte SecurityEnable );
```

Parameter Details

`TranSeq` – The transaction sequence number.

`DstAddr` – The destination address.

`Status` – defined in `ZComDef.h` as `ZStatus_t`.

`RoutingTableEntries` – The number of possible items in this message.

`StartIndex` – the responding device could have more response items than can fit in a response message, the requester can specify a starting index into the possible response items. This field is the starting index for this response message.

`RoutingListCount` – The number of response items in this message.

`RoutingTableList` – the list of routing records. You can look up `rtgItem_t` in `ZDProfile.h` for information on this structure.

`SecurityEnable` – true if security is enabled.

Return

`ZStatus_t` – status values defined in `ZStatus_t` in `ZComDef.h`.

3.2.6.7 ZDP_MgmtBindReq ()

If the device supports this command (optional), calling this function will generate the request for the destination device to return its Binding Table. The calling application can only call this function if the `ZDO_MGMT_BIND_REQUEST` compile flag is set either in `ZDConfig.h` or as a normal compile flag.

Prototype

```
afStatus_t ZDP_MgmtBindReq( zAddrType_t *dstAddr, byte StartIndex, byte SecurityEnable );
```

Parameter Details

`DstAddr` - The destination address.

`StartIndex` – the responding device could have more response items than can fit in a response message, the requester can specify a starting index into the possible response items.

`SecurityEnable` – true if security is enabled.

Return

`afStatus_t` – This function uses AF to send the message, so the status values are AF status values defined in `ZStatus_t` in `ZComDef.h`.

3.2.6.8 ZDP_MgmtBindRsp()

If the device supports this command (optional), calling this function will generate the response. The ZDO will generate this message automatically when a “Management Binding Request” message is received if the `ZDO_MGMT_BIND_RESPONSE` compile flag is set either in `ZDConfig.h` or as a normal compile flag.

Prototype

```
ZStatus_t ZDP_MgmtBindRsp( byte TranSeq, zAddrType_t *dstAddr,
                           byte Status, byte BindingTableEntries,
                           byte StartIndex, byte BindingTableListCount,
                           apsBindingItem_t *BindingTableList,
                           byte SecurityEnable );
```

Parameter Details

TranSeq – The transaction sequence number.

DstAddr - The destination address.

Status – defined in ZComDef.h as ZStatus_t.

BindingTableEntries – The number of possible items in this message.

StartIndex – the responding device could have more response items than can fit in a response message, the requester can specify a starting index into the possible response items. This field is the starting index for this response message.

BindingTableListCount – The number of response items in this message.

BindingTableList – the list of Binding table records. You can look up apsBindingItem_t in APSMEDE.h for information on this structure.

SecurityEnable – true if security is enabled.

Return

ZStatus_t –status values defined in ZStatus_t in ZComDef.h.

3.2.6.9 ZDP_MgmtLeaveReq ()

If the device supports this command (optional), calling this function will generate the request for the destination device to leave the network or request another device to leave. The calling application can only call this function if the ZDO_MGMT_LEAVE_REQUEST compile flag is set either in ZDConfig.h or as a normal compile flag.

Prototype

```
afStatus_t ZDP_MgmtLeaveReq( zAddrType_t *dstAddr, byte *IEEEAddr, byte SecurityEnable );
```

Parameter Details

DstAddr - The destination address.

IEEEAddr – 64 bit address (8bytes) of device to leave

SecurityEnable – true if security is enabled.

Return

afStatus_t – This function uses AF to send the message, so the status values are AF status values defined in ZStatus_t in ZComDef.h.

3.2.6.10 ZDP_MgmtLeaveRsp()

If the device supports this command (optional), calling this function will generate the response. The ZDO will generate this message automatically when a “Management Leave Request” message is received if the ZDO_MGMT_LEAVE_RESPONSE compile flag is set either in ZDConfig.h or as a normal compile flag.

Prototype

```
ZStatus_t ZDP_MgmtLeaveRsp( byte TranSeq, zAddrType_t *dstAddr,
                           byte Status, byte SecurityEnable );
```

Parameter Details

TranSeq - The transaction sequence number.
 DstAddr - The destination address.
 Status – defined in ZComDef.h as ZStatus_t.
 SecurityEnable – true if security is enabled.

Return

ZStatus_t –status values defined in ZStatus_t in ZComDef.h.

3.2.6.11 ZDP_MgmtDirectJoinReq ()

If the device supports this command (optional), calling this function will generate the request for the destination device to direct join another device. The calling application can only call this function if the ZDO_MGMT_JOINDIRECT_REQUEST compile flag is set either in ZDConfig.h or as a normal compile flag.

Prototype

```
afStatus_t ZDP_MgmtDirectJoinReq( zAddrType_t *dstAddr,
                                   byte *deviceAddr, byte capInfo, byte SecurityEnable );
```

Parameter Details

DstAddr - The destination address.
 deviceAddr – 64 bit address (8bytes) of device to Join.
 capInfo – Capability information of the device to join.

Types	Bit
CAPINFO_ALTPANCOORD	0x01
CAPINFO_DEVICETYPE_FFD	0x02
CAPINFO_POWER_AC	0x04
CAPINFO_RCVR_ON_IDLE	0x08
CAPINFO_SECURITY_CAPABLE	0x40
CAPINFO_ALLOC_ADDR	0x80

SecurityEnable – true if security is enabled.

Return

`afStatus_t` – This function uses AF to send the message, so the status values are AF status values defined in `ZStatus_t` in `ZComDef.h`.

3.2.6.12 ZDP_MgmtDirectJoinRsp()

If the device supports this command (optional), calling this function will generate the response. The ZDO will generate this message automatically when a “Management Direct Join Request” message is received if the `ZDO_MGMT_JOINDIRECT_RESPONSE` compile flag is set either in `ZDConfig.h` or as a normal compile flag.

Prototype

```
ZStatus_t ZDP_MgmtDirectJoinRsp( byte TranSeq, zAddrType_t *dstAddr, byte Status, byte SecurityEnable );
```

Parameter Details

`TranSeq` – The transaction sequence number.

`dstAddr` – The destination address.

`Status` – defined in `ZComDef.h` as `ZStatus_t`.

`SecurityEnable` – true if security is enabled.

Return

`ZStatus_t` – status values defined in `ZStatus_t` in `ZComDef.h`.

3.2.6.13 ZDP_MgmtPermitJoinReq()

This is a macro that calls `ZDP_SendData()` directly. The function builds and sends `Mgmt_Permit_Joining_req` to request a remote device to allow or disallow association. The request is normally generated by a commissioning tool or network management device. Additionally, if field `TC_Significance` is set to `0x01` and the remote device is the trust center, the trust center authentication policy will be affected. The detailed procedure upon receipt of `Mgmt_Permit_Joining_req` can be found in the latest ZigBee Specification Section 2.4.3.3.7.

Prototype

```
afStatus_t ZDP_MgmtPermitJoinReq( zAddrType_t *dstAddr, byte duration,  
                                byte TcSignificance, byte SecurityEnable );
```

Parameter Details

`dstAddr` – The destination address.

`duration` – The length of time in seconds during which ZigBee coordinator or router will allow association. The value `0x00` and `0xff` indicate that permission is disabled or enabled, respectively, without a specified time limit.

`TcSignificance` – This is a boolean value. If it is set to `0x01` and the remote device is the trust center, the command affects the trust center authentication policy as described in ZigBee Specification Section 2.4.3.3.7.2.

`SecurityEnable` – This field is a Boolean value. It is set to true if security is enabled.

Return

`ZStatus_t` –status values defined in `ZStatus_t` in `ZComDef.h`.

3.2.6.14 ZDP_MgmtPermitJoinRsp()

This is a macro that calls `ZDP_SendData ()` directly. If the device supports this command (optional), calling this function will generate the response for `Mgmt_Permit_Joining_req` request. The ZDO will generate this message automatically when an `Mgmt_Permit_Joining_req` message is received if the `ZDO_MGMT_PERMIT_JOIN_RESPONSE` compile flag is set either in `ZDConfig.h` or as a normal compile flag.

Prototype

```
ZStatus_t ZDP_MgmtPermitJoinRsp( byte *TransSeq,
                                zAddrType_t *dstAddr, byte *Statue, byte SecurityEnable );
```

Parameter Details

`TransSeq` – The transaction sequence number.

`dstAddr` – The destination address.

`Status` – The status of the response. It is either `SUCCESS` or `INVALID_REQUEST`.

`SecurityEnable` – This field is a Boolean value. It is set to true if security is enabled.

Return

`ZStatus_t` –status values defined in `ZStatus_t` in `ZComDef.h`.

3.2.6.15 ZDP_MgmtNwkUpdateReq()

This function builds and sends an `Mgmt_NWK_Update_req` message. This command is provided to allow updating of network configuration parameters or to request information from devices on network conditions in the local operating environment. This function sends a unicast or broadcast message.

Prototype

```
afStatus_t ZDP_MgmtNwkUpdateReq( zAddrType_t *dstAddr, uint32 ChannelMask,
                                uint8 ScanDuration, uint8 ScanCount,
                                uint8 NwkUpdateId, uint16 NwkManagerAddr );
```

Parameter Details

`dstAddr` – The destination address of the message.

`ChannelMask` – The 32-bit channel bitmap.

`ScanDuration` – A value used to calculate the length of time to spend scanning each channel. If `ScanCount` has a value of `0xfe` this is a request for channel change. If `ScanCount` has a value of `0xff` this is a request to change the *apsChannelMask* and *nwkManagerAddr* parameters.

ScanCount – The number of energy scans to be conducted and reported. This parameter is used only if the **ScanDuration** is within the range of 0x00 to 0x05.

NwkUpdateId – The network update id value. This parameter is used only if the **ScanDuration** is 0xfe or 0xff.

NwkManagerAddr – The NWK address for the device with the Network Manager bit set in its Node Descriptor. This parameter is used only if the **ScanDuration** is set to 0xff.

Return

ZStatus_t –status values defined in **ZStatus_t** in **ZComDef.h**.

3.2.6.16 ZDP_MgmtNwkUpdateNotify()

This function builds and sends an **Mgmt_NWK_Update_notify** message. The **Mgmt_NWK_Update_notify** is provided to enable ZigBee devices to report the condition on local channels to a network manager. This function sends a unicast message.

Prototype

```
ZStatus_t ZDP_MgmtNwkUpdateNotify( byte *TransSeq, zAddrType_t *dstAddr,
                                     uint8 status, uint32 scannedChannels,
                                     uint16 totalTransmissions, uint16 transmissionFailures,
                                     uint8 listCount, uint8 *energyValues,
                                     uint8 txOptions, byte SecurityEnable );
```

Parameter Details

TransSeq – The transaction sequence number.

dstAddr – The destination address of the message.

status – The status of the notify command.

scannedChannels – The list of channels scanned by the request.

totalTransmissions – The count of the total transmissions reported by the device .

transmissionFailures – The sum of the total transmission failures reported by the device.

listCount – The number of the records contained in the **energyValues** parameter.

energyValues – The result of an energy measurement made on the scanned channels.

txOptions – The transmit options.

SecurityEnable – This field is a Boolean value. It is set to true if security is enabled.

Return

ZStatus_t –status values defined in **ZStatus_t** in **ZComDef.h**.

3.2.7 ZDO Parsing Functions

These functions are used to parse incoming messages (usually response messages).

3.2.7.1 ZDO_ParseAddrRsp

Parse the NWK_addr_rsp and IEEE_addr_rsp messages

Prototype

```
ZDO_NwkIEEEAddrResp_t *ZDO_ParseAddrRsp( zdoIncomingMsg_t *inMsg );
```

Parameter Details

inMsg – Pointer to the incoming (raw) message.

Return

ZDO_NwkIEEEAddrResp_t – a pointer to parsed structures. This structure was allocated using osal_mem_alloc, so it must be freed by the calling function [osal_mem_free()].

3.2.7.2 ZDO_ParseNodeDescRsp

Parse the Node_Desc_rsp message.

Prototype

```
void ZDO_ParseNodeDescRsp( zdoIncomingMsg_t *inMsg,  
                           ZDO_NodeDescRsp_t *pNDRsp );
```

Parameter Details

inMsg – Pointer to the incoming (raw) message.

pNDRsp – A place to parse the message into.

Return

None.

3.2.7.3 ZDO_ParsePowerDescRsp

Parse the Power_Desc_rsp message.

Prototype

```
void ZDO_ParsePowerDescRsp( zdoIncomingMsg_t *inMsg,  
                           ZDO_PowerRsp_t *pNPRsp );
```

Parameter Details

inMsg – Pointer to the incoming (raw) message.

pNPRsp – A place to parse the message into.

Return

None.

3.2.7.4 ZDO_ParseSimpleDescRsp

Parse the Simple_Desc_rsp message.

Prototype

```
void ZDO_ParseSimpleDescRsp( zdoIncomingMsg_t *inMsg,  
                             ZDO_SimpleDescRsp_t *pSimpleDescRsp );
```

Parameter Details

inMsg – Pointer to the incoming (raw) message.

pSimpleDescRsp – A place to parse the message into. The pAppInClusterList and pAppOutClusterList fields in the SimpleDescriptionFormat_t structure are allocated and the calling function needs to free [osal_msg_free()] these buffers.

Return

None.

3.2.7.5 ZDO_ParseEPListRsp

Parse the Active_EP_rsp or Match_Desc_rsp message.

Prototype

```
ZDO_ActiveEndpointRsp_t *ZDO_ParseEPListRsp( zdoIncomingMsg_t *inMsg );
```

Parameter Details

inMsg – Pointer to the incoming (raw) message.

Return

ZDO_ActiveEndpointRsp_t – a pointer to parsed structures. This structure was allocated using osal_mem_alloc, so it must be freed by the calling function [osal_mem_free()].

3.2.7.6 ZDO_ParseBindRsp

Parse the Bind_rsp, Unbind_rsp or End_Device_Bind_rsp message.

Prototype

```
#define ZDO_ParseBindRsp(a) ((uint8)(*(a->asdu)))
```

Parameter Details

a – pointer to the message to parse [zdoIncomingMsg_t *]

Return

uint8 – status field of the message.

3.2.7.7 ZDO_ParseMgmNwkDiscRsp

Parse the Mgmt_NWK_Disc_rsp message.

Prototype

```
ZDO_MgmNwkDiscRsp_t *ZDO_ParseMgmNwkDiscRsp( zdoIncomingMsg_t *inMsg );
```

Parameter Details

inMsg – Pointer to the incoming (raw) message.

Return

ZDO_MgmNwkDiscRsp_t – a pointer to parsed structures (NULL if not allocated). This structure was allocated using osal_mem_alloc, so it must be freed by the calling function [osal_mem_free()].

3.2.7.8 ZDO_ParseMgmtLqiRsp

Parse the Mgmt_Lqi_rsp message.

Prototype

```
ZDO_MgmtLqiRsp_t *ZDO_ParseMgmtLqiRsp( zdoIncomingMsg_t *inMsg );
```

Parameter Details

inMsg – Pointer to the incoming (raw) message.

Return

ZDO_MgmtLqiRsp_t – a pointer to parsed structures (NULL if not allocated). This structure was allocated using osal_mem_alloc, so it must be freed by the calling function [osal_mem_free()].

3.2.7.9 ZDO_ParseMgmtRtgRsp

Parse the Mgmt_Rtg_rsp message.

Prototype

```
ZDO_MgmtRtgRsp_t *ZDO_ParseMgmtRtgRsp( zdoIncomingMsg_t *inMsg );
```

Parameter Details

`inMsg` – Pointer to the incoming (raw) message.

Return

`ZDO_MgmtRtgRsp_t` – a pointer to parsed structures (NULL if not allocated). This structure was allocated using `osal_mem_alloc`, so it must be freed by the calling function [`osal_mem_free()`].

3.2.7.10 ZDO_ParseMgmtBindRsp

Parse the `Mgmt_Bind_rsp` message.

Prototype

```
ZDO_MgmtBindRsp_t *ZDO_ParseMgmtBindRsp( zdoIncomingMsg_t *inMsg );
```

Parameter Details

`inMsg` – Pointer to the incoming (raw) message.

Return

`ZDO_MgmtBindRsp_t` – a pointer to parsed structures (NULL if not allocated). This structure was allocated using `osal_mem_alloc`, so it must be freed by the calling function [`osal_mem_free()`].

3.2.7.11 ZDO_ParseMgmtDirectJoinRsp

Parse the `Mgmt_Direct_Join_rsp` message.

Prototype

```
#define ZDO_ParseMgmtDirectJoinRsp(a) (((uint8)(*(a->asdu)))
```

Parameter Details

`a` – pointer to the message to parse [`zdoIncomingMsg_t *`]

Return

`uint8` – status field of the message.

3.2.7.12 ZDO_ParseMgmtLeaveRsp

Parse the `Mgmt_Leave_rsp` message.

Prototype

```
#define ZDO_ParseMgmtLeaveRsp(a) (((uint8)(*(a->asdu)))
```

Parameter Details

a – pointer to the message to parse [zdoIncomingMsg_t *]

Return

uint8 – status field of the message.

3.2.7.13 ZDO_ParseMgmtPermitJoinRsp

Parse the Mgmt_Permit_Join_rsp message.

Prototype

```
#define ZDO_ParseMgmtPermitJoinRsp(a) ((uint8)(*(a->asdu)))
```

Parameter Details

a – pointer to the message to parse [zdoIncomingMsg_t *]

Return

uint8 – status field of the message.

3.2.7.14 ZDO_ParseUserDescRsp

Parse the User_Desc_rsp message.

Prototype

```
ZDO_UserDescRsp_t *ZDO_ParseUserDescRsp( zdoIncomingMsg_t *inMsg );
```

Parameter Details

inMsg – Pointer to the incoming (raw) message.

Return

ZDO_UserDescRsp_t – a pointer to parsed structures (NULL if not allocated). This structure was allocated using osal_mem_alloc, so it must be freed by the calling function [osal_mem_free()].

3.2.7.15 ZDO_ParseServerDiscRsp

Parse the Server_Discovery_rsp message.

Prototype

```
void ZDO_ParseServerDiscRsp( zdoIncomingMsg_t *inMsg,  
ZDO_ServerDiscRsp_t *pRsp );
```

Parameter Details

inMsg – Pointer to the incoming (raw) message.

pRsp – A place to parse the message into.

Return

None.

3.2.7.16 ZDO_ParseEndDeviceBindReq

Parse the End_Device_Bind_req message message.

Prototype

```
void ZDO_ParseEndDeviceBindReq( zdoIncomingMsg_t *inMsg,  
ZDEndDeviceBind_t *bindReq );
```

Parameter Details

inMsg – Pointer to the incoming (raw) message.

bindReq – A place to parse the message into. NOTE: The input and output cluster lists were allocated using osal_mem_alloc, so they must be freed by the calling function [osal_mem_free()].

Return

None.

3.2.7.17 ZDO_ParseBindUnbindReq

Parses the Bind_req or Unbind_req messages.

Prototype

```
void ZDO_ParseBindUnbindReq( zdoIncomingMsg_t *inMsg,  
ZDO_BindUnbindReq_t *pReq );
```

Parameter Details

inMsg – Pointer to the incoming (raw) message.

pReq – A place to parse the message into.

Return

None.

3.2.7.18 ZDO_ParseUserDescConf

Parse the User_Desc_conf message.

Prototype

```
#define ZDO_ParseUserDescConf(a) ((uint8)((a->asdu)))
```

Parameter Details

a – pointer to the message to parse [zdoIncomingMsg_t *]

Return

uint8 – status field of the message.

3.2.7.19 ZDO_ParseDeviceAnnce

Called to parse an End_Device_annce message.

Prototype

```
void ZDO_ParseDeviceAnnce( zdoIncomingMsg_t *inMsg,  
                           ZDO_DeviceAnnce_t *pAnnce );
```

Parameter Details

inMsg – Pointer to the incoming (raw) message.

pAnnce – A place to parse the message into.

Return

None.

3.2.7.20 ZDO_ParseMgmtNwkUpdateNotify

Parse the Mgmt_NWK_Update_notify message.

Prototype

```
ZDO_MgmtNwkUpdateNotify_t *ZDO_ParseMgmtNwkUpdateNotify(  
    zdoIncomingMsg_t *inMsg );
```

Parameter Details

inMsg – Pointer to the incoming (raw) message.

Return

ZDO_MgmtNwkUpdateNotify_t - a pointer to parsed structures (NULL if not allocated). This structure was allocated using osal_mem_alloc, so it must be freed by the calling function [osal_mem_free()].

3.2.8 ZDO Network Manager

The Network Manager controls and monitors frequency agility and PAN ID conflicts. For a better description please see the ZStack Developer's Guide.

3.2.8.1 NwkMgr_SetNwkManager

In a given network, there is only one Zigbee Network Manager device that controls network wide frequency agility and PAN ID conflict (default is the coordinator). Every device has a local ZDO network manager to participate in frequency agility and PAN ID conflict. To set a device (Coordinator or router only) as the Zigbee Network Manager, the application can call this function after it joins the network (must include the NWK_MANAGER compile flag).

Prototype

```
void NwkMgr_SetNwkManager( void );
```

Parameter Details

None.

Return

None.

3.3 Application Framework (AF)

The Application Framework layer is the application's over-the-air data interface to the APS layer. It contains the functions an application uses to send data out over the air (through the APS and NWK) layers. This layer is also the endpoint multiplexer for incoming data messages.

3.3.1 Overview

The AF provides the following functionality to applications:

- Endpoint Management
- Sending and Receiving Data

3.3.1.1 Endpoint Management

Each device is a node in the Zigbee. Each node has a long and short address, the short address of the node is used by other nodes to send it data. Each node has 241 endpoint (0 reserved, 1-240 application assigned). Each endpoint is separately addressable; when a device sends data it must specify the destination node's short address and the endpoint that will receive that data.

An application must register one or more endpoints to send and receive data in a Zigbee network.

3.3.1.1.1 Simple Descriptor - SimpleDescriptionFormat_t

Each endpoint must have a Zigbee Simple Descriptor. This descriptor describes the endpoint to the rest of the Zigbee network. Another device can query and endpoint for its simple descriptor and know what kind of device it is. This structure is defined by the application.

```
typedef struct
{
    byte      EndPoint;
    uint16    AppProfId;
    uint16    AppDeviceId;
    byte      AppDevVer:4;
    byte      Reserved:4;           // AF_V1_SUPPORT uses for AppFlags:4.
    byte      AppNumInClusters;
    cId_t     *pAppInClusterList;
    byte      AppNumOutClusters;
    cId_t     *pAppOutClusterList;
} SimpleDescriptionFormat_t;
```

EndPoint - The endpoint number 1-240 (0 is reserved). This is the subaddress of the node, and is used to receive data..

AppProfId - This field identifies the Profile ID supported on this endpoint. The IDs shall be obtained from the ZigBee Alliance.

AppDeviceId - This field identifies the Device ID supported on this endpoint. The IDs shall be obtained from the ZigBee Alliance.

AppDevVer - Identifies the version of the relevant Device Description that this device implements on this endpoint. 0x00 is Version 1.0.

Reserved - not used.

AppNumInClusters - This indicates the number of input clusters supported by this endpoint.

pAppInClusterList - Pointer to the input cluster ID list.

AppNumOutClusters - 8 bit field. This indicates the number of output clusters supported by this endpoint.

pAppOutClusterList - Pointer to the output cluster ID list.

3.3.1.1.2 Endpoint Descriptor - endPointDesc_t

This structure is the endpoint descriptor. For every endpoint wanted/needed in the node there must be one endpoint descriptor (usually defined in the application).

```
typedef struct
{
    byte endPoint;
    byte *task_id; // Pointer to location of the Application task ID.
    SimpleDescriptionFormat_t *simpleDesc;
    afNetworkLatencyReq_t latencyReq;
} endPointDesc_t;
```

endPoint - The endpoint number 1-240 (0 is reserved). This is the subaddress of the node, and is used to receive data..

task_id - Task ID pointer. When a message is received, this task ID will be used to determine where the message is delivered. The received message is packaged as an OSAL message and sent to the task (command ID = AF_INCOMING_MSG_CMD).

simpleDesc - Pointer to the Zigbee Simple Descriptor for this endpoint.

latencyReq - This field must be filled with noLatencyReqs

3.3.1.1.3 afRegister()

This function is used to register a new endpoint for the device. The application will call this function for each endpoint that the application. This function doesn't allow duplicate endpoint registration.

Prototype

```
afStatus_t afRegister( endPointDesc_t *epDesc );
```

Parameter Details

epDesc – Pointer to the Endpoint descriptor (defined above).

Return

afStatus_t – ZSuccess if successful (defined in ZComDef.h). Errors are defined in ZComDef.h.

3.3.1.1.4 afRegisterExtended()

This function serves the same function as afRegister() [to register an endpoint], but this function specifies a callback function to be called when the endpoint's simple descriptor is queried. This will allow an application to dynamically change the simple descriptor and not use the RAM/ROM needed to store the descriptor.

Prototype

```
epList_t *afRegisterExtended( endPointDesc_t *epDesc, pDescCB descFn );
```

Parameter Details

epDesc – Pointer to the Endpoint descriptor (defined above).

descFn – Callback function pointer. This function will be called when the endpoint's simple descriptor is queried. The referenced function must allocate enough space for a simple descriptor, fill in the simple descriptor then return the pointer to the simple descriptor. The caller will free the descriptor.

Return

epList * – Pointer to the endpoint list item. NULL if not successful.

3.3.1.1.5 afFindEndPointDesc()

Use this function to find an endpoint descriptor from an endpoint.

Prototype

```
endPointDesc_t *afFindEndPointDesc( byte endPoint );
```

Parameter Details

endPoint – The endpoint number of the endpoint descriptor you are looking for.

Return

endPointDesc_t * – Pointer to the endpoint descriptor. NULL if not successful.

3.3.1.1.6 afFindSimpleDesc()

Use this function to find an endpoint descriptor from an endpoint.

Prototype

```
byte afFindSimpleDesc( SimpleDescriptionFormat_t **ppDesc, byte EP );
```

Parameter Details

ppDesc – Pointer to a pointer of the simple descriptor. This space may have been allocated. The return from this function will tell if the memory was allocated and needs to be freed [osal_mem_free()].

EP – endpoint of simple descriptor needed.

Return

If return value is not zero, the descriptor's memory must be freed by the caller [osal_mem_free()].

3.3.1.1.7 afGetMatch()

By default, the device will respond to the ZDO Match Descriptor Request. You can use this function to get the setting for the ZDO Match Descriptor Response.

Prototype

```
uint8 afGetMatch( uint8 ep );
```

Parameter Details

ep – the endpoint to get the ZDO Match Descriptor Response behavior.

Return

true means allows responses, false means not allowed or endpoint not found.

3.3.1.1.8 afSetMatch()

By default, the device will respond to the ZDO Match Descriptor. You can use this function to change this behavior. For example, if the endpoint parameter is 1 and action is false, the ZDO will not respond to a ZDO Match Descriptor Request for endpoint 1.

Prototype

```
uint8 afSetMatch( uint8 ep, uint8 action );
```

Parameter Details

ep – the endpoint to change the ZDO Match Descriptor Response behavior.

action – true is to allow responses (default), and false is to not allow responses for this ep.

Return

true if success, false if endpoint not found.

3.3.1.1.9 afNumEndPoints()

Use this function to lookup the number of endpoints registered.

Prototype

```
byte afNumEndPoints( void );
```

Parameter Details

None.

Return

Number of endpoints registered for this device (including endpoint 0 – reserved for ZDO).

3.3.1.1.10 afEndPoints ()

This function will return an array of endpoints registered. It fills in the input buffer (`epBuf`) with the endpoint (numbers). Use `afNumEndPoints()` to find out how big a buffer to supply in `epBuf`. For example, if your application registered 2 endpoints (endpoint 10 and endpoint 15) and `skipZDO` is true, the array (`epBuf`) will contain 2 bytes (10 & 15).

Prototype

```
void afEndPoints( byte *epBuf, byte skipZDO );
```

Parameter Details

`epBuf` – pointer to buffer to fill in the endpoint numbers. There will be one byte for each endpoint.

`skipZDO` – true to not include the ZDO endpoint (0).

Return

None.

3.3.1.1.11 afAPSF_ConfigGet ()

This function will ascertain the APS Fragmentation parameters for the specified EndPoint; it must be called after first registering the EndPoint. This function is mainly intended for use by the APSF library module.

Prototype

```
void afAPSF_ConfigGet(uint8 endPoint, afAPSF_Config_t *pCfg);
```

Parameter Details

`endPoint` – The specific EndPoint for which to get the fragmentation configuration.

pCfg – A pointer to an APSF configuration structure to fill with values.

Return

None.

3.3.1.1.12 afAPSF_ConfigSet ()

This function will attempt to set the APS Fragmentation parameters for the specified EndPoint; it must be called after first registering the EndPoint.

Prototype

```
afStatus_t afAPSF_ConfigSet(uint8 endPoint, afAPSF_Config_t *pCfg);
```

Parameter Details

endPoint – The specific EndPoint for which to set the fragmentation configuration.

pCfg – A pointer to an APSF configuration structure to filled with values.

Return

afStatus_SUCCESS for success; afStatus_INVALID_PARAMETER if the specified EndPoint is not registered.

3.3.1.2 Sending Data

3.3.1.2.1 AF_DataRequest()

Call this function to send data.

Prototype

```
afStatus_t AF_DataRequest( afAddrType_t *dstAddr, endPointDesc_t *srcEP,
                          uint16 cID, uint16 len, uint8 *buf, uint8 *transID,
                          uint8 options, uint8 radius );
```

Parameter Details

dstAddr – Destination address pointer. The address mode in this structure must be either: afAddrNotPresent to let the reflector (source binding) figure out the destination address; afAddrGroup to send to a group; afAddrBroadcast to send a broadcast message; or afAddr16Bit to send directly (unicast) to a node.

srcEP – Endpoint Descriptor pointer of the sending endpoint.

cID – Cluster ID – the message's cluster ID is like a message ID and is unique with in the profile.

len – number of bytes in the buf field. The number of bytes to send.

buf – buffer pointer of data to send.

transID – transaction sequence number pointer. This number will be incremented by this function if the message is buffered to be sent.

options – the options to send this message are to be OR'd into this field are:

Name	Value	Description
AF_WILDCARD_PROFILEID	0x02	Will force the message to use Wildcard ProfileID
AF_ACK_REQUEST	0x10	APS Ack requested. This is an application level acknowledgement – meaning that the destination device will acknowledge the message. Only used on messages send direct (unicast).
AF_SUPRESS_ROUTE_DISC_NETWORK	0x20	Supress Route Discovery for intermediate routes
AF_EN_SECURITY	0x40	Not needed. Reference ZStack Developer’s Guide for security.
AF_SKIP_ROUTING	0x80	Setting this option will cause the device to skip routing and try to send the message directly (not multihop). End devices will not send the message to its parent first. Good for only direct (unicast) and broadcast messages.
AF_DISCV_ROUTE	0x00	This option is no longer available and is include for backwards compatibility

radius – Maximum number of hops. Use AF_DEFAULT_RADIUS as the default.

Return

afStatus_t – ZSuccess if successful (defined in ZComDef.h). Errors are defined in ZComDef.h.

3.3.1.2.2 afDataReqMTU()

Use this function to find the maximum number of data bytes that can be sent based on the input parameters.

Prototype

```
uint8 afDataReqMTU( afDataReqMTU_t* fields );
```

Parameter Details

fields – parameters for the type of message to be sent. The fields are:

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

kvp – set this to false.

```
typedef struct
{
    uint8 secure;
} APSDE_DataReqMTU_t;
```

aps.secure – set this to false. It will automatically be set if you are in a secure network.

Return

The maximum number of bytes that can be sent.

3.3.1.3 Receiving Data

Data packets are sent to an application's registered endpoint. The application will receive the `AF_INCOMING_MSG_CMD` OSAL message in its OSAL event processing function (all sample applications have an example of receiving these messages). The message comes in the following AF structure:

```
typedef struct
{
    osal_event_hdr_t hdr; /* OSAL Message header */
    uint16 groupId; /* Message's group ID - 0 if not set */
    uint16 clusterId; /* Message's cluster ID */
    afAddrType_t srcAddr; /* Source Address, if endpoint is STUBAPS_INTER_PAN_EP,
                           it's an InterPAN message */
    uint16 macDestAddr; /* MAC header destination short address */
    uint8 endPoint; /* destination endpoint */
    uint8 wasBroadcast; /* TRUE if network destination was a broadcast address */
    uint8 LinkQuality; /* The link quality of the received data frame */
    uint8 correlation; /* The raw correlation value of the received data frame */
    int8 rssi; /* The received RF power in units dBm */
    uint8 SecurityUse; /* deprecated */
    uint32 timestamp; /* receipt timestamp from MAC */
    afMSGCommandFormat_t cmd; /* Application Data */
} afIncomingMSGPacket_t;
```

3.4 Application Support Sub-Layer (APS)

3.4.1 Overview

The APS provides the following management functionality accessible to the higher layers:

- Binding Table Management
- Group Table Management
- Quick Address Lookup

Besides the management functions, APS also provides data services that are not accessible to the application. Applications should send data through the AF data interface [`AF_DataRequest()`]. To use the binding table functions include "BindingTable.h" in your application.

3.4.2 Binding Table Management

The APS Binding table is a table defined in static RAM (not allocated). The table size can be controlled by 2 configuration items in `f8wConfig.cfg` [`NWK_MAX_BINDING_ENTRIES` and `MAX_BINDING_CLUSTER_IDS`]. `NWK_MAX_BINDING_ENTRIES` defines the number of entries in the table and `MAX_BINDING_CLUSTER_IDS` defines the number of clusters (16 bits each) in each entry. The table is defined in `nwk_globals.c`.

The table is only included (along with the following functions) if `REFLECTOR` or `COORDINATOR_BINDING` is defined. Using the compiler directive `REFLECTOR` enables the source binding features in the APS layer.

3.4.2.1 Binding Record Structure – `BindingEntry_t`

```
typedef struct
{
    // No src address since the src is always the local device
    uint8 srcEP;
    uint8 dstGroupMode; // Destination address type; 0 - Normal address index,
                        // 1 - Group address
    uint16 dstIdx;      // This field is used in both modes (group and non-group)
                        // to save NV and RAM space
                        // dstGroupMode = 0 - Address Manager index
                        // dstGroupMode = 1 - Group Address
    uint8 dstEP;
    uint8 numClusterIds;
    uint16 clusterIdList[MAX_BINDING_CLUSTER_IDS];
                        // Don't use MAX_BINDING_CLUSTERS_ID when
                        // using the clusterIdList field. Use
                        // gMAX_BINDING_CLUSTER_IDS
} BindingEntry_t;
```

`srcEP` – source endpoint.

`dstGroupMode` – destination address type. If this field contains a 0, the destination address is a normal address. If this field contains a 1, the `dstIdx` is the destination group address.

`dstIdx` – If `dstGroupMode` contains a 0, this field contains an address manager index for the destination address. If `dstGroupMode` contains a 1, this field contains destination group address.

`dstEP` – destination endpoint.

`numClusterIds` – number of entries in `clusterIdList`.

`clusterIdList` – Cluster ID list. The maximum number of cluster IDs stored in this array is defined with `MAX_BINDING_CLUSTER_IDS` [defined in `f8wConfig.cfg`].

3.4.2.2 Binding Table Maintenance

3.4.2.2.1 `bindAddEntry()`

Use this function to add an entry into the binding table. Since each table entry can have multiple cluster IDs, this function may just add a cluster ID(s) to an existing binding table record.

Prototype

```
BindingEntry_t *bindAddEntry( byte srcEp,
                             zAddrType_t *dstAddr,
                             byte dstEp,
                             byte numClusterIds,
                             uint16 *clusterIds );
```

Parameter Details

srcEp – binding record source endpoint.

dstAddr – binding record destination address. This address type must contain either Addr16Bit, Addr64Bit or AddrGroup addrMode with the appropriate addr field filled in. If the addrMode is AddrGroup, the group ID is put in the addr.shortAddr field.

dstEp – binding record destination endpoint. This field is N/A if the dstAddr is a group address.

numClusterIds – Number of cluster IDs in the clusterIds.

clusterIds – Pointer to a list of cluster IDs to add. This points to a list (numClusterIds) of 16-bit cluster IDs.

Return

BindingEntry_t * - Pointer to the newly added binding entry. NULL if not added.

3.4.2.2.2 bindRemoveEntry()

Remove a binding entry from the binding table.

Prototype

```
byte bindRemoveEntry( BindingEntry_t *pBind );
```

Parameter Details

pBind – pointer to the binding entry to delete from the binding table.

Return

true – this function doesn't check for a valid entry.

3.4.2.2.3 bindRemoveClusterIdFromList()

This function removes a cluster ID from the cluster ID list of an existing binding table entry. This function assumes that the passed in entry is valid.

Prototype

```
byte bindRemoveClusterIdFromList( BindingEntry_t *entry, uint16 clusterId );
```

Parameter Details

entry – pointer to a valid binding table entry.

clusterId – 16 bit cluster ID to delete from the cluster list.

Return

true if at least 1 cluster ID is left in the list after the delete, false if cluster list is empty. If the cluster list is empty it should be deleted by the calling function.

3.4.2.2.4 bindAddClusterIdToList()

This function will add a cluster ID to the cluster ID list of an existing binding table entry. This function assumes that the passed in entry is valid.

Prototype

```
byte bindAddClusterIdToList( BindingEntry_t *entry, uint16 clusterId );
```

Parameter Details

entry – pointer to a valid binding table entry.

clusterId – 16 bit cluster ID to add to the cluster list.

Return

true if added to the list, false if not added. false means that either the entry is NULL or the number of cluster IDs is already at it's maximum (MAX_BINDING_CLUSTER_IDS).

3.4.2.2.5 bindRemoveDev()

Use this function to remove all binding table entries with the passed in address. If the address matches either source or destination address in the binding record the entry will be removed.

Prototype

```
void bindRemoveDev( zAddrType_t *Addr );
```

Parameter Details

Addr – address to remove from the binding table. Works for addrMode equal to either Addr16Bit, Addr64Bit or AddrGroup.

Return

None.

3.4.2.2.6 bindRemoveSrcDev()

Use this function to remove all binding table entries with the passed in source address and endpoint.

Prototype

```
void bindRemoveSrcDev( zAddrType_t *srcAddr, uint8 ep );
```

Parameter Details

srcAddr – address to remove from the binding table. Works for addrMode equal to either Addr16Bit or Addr64Bit.

ep – source endpoint.

Return

None.

3.4.2.2.7 bindUpdateAddr ()

Use this function to count exchange short addresses in the binding table. All entries with the oldAddr will be replaced with newAddr.

Prototype

```
void bindUpdateAddr( uint16 oldAddr, uint16 newAddr );
```

Parameter Details

oldAddr – old short (network) address to search for.

newAddr – short (network) address to replace oldAddr with.

Return

None.

3.4.2.3 Binding Table Searching**3.4.2.3.1 bindFindExisting ()**

Find an existing binding table entry. Using a source address and endpoint and a destination address and endpoint to search the binding table.

Prototype

```
BindingEntry_t *bindFindExisting( byte srcEp, zAddrType_t *dstAddr, byte dstEp );
```

Parameter Details

srcEp – binding record source endpoint.

dstAddr – binding record destination address. This address type must contain either Addr16Bit, Addr64Bit or AddrGroup addrMode with the appropriate addr field filled in. If the addrMode is AddrGroup, the group ID is put in the addr.shortAddr field.

dstEp – binding record destination endpoint. This field is N/A if the dstAddr is a group address.

Return

BindingEntry_t * - Pointer to the found binding entry. NULL if not found.

3.4.2.3.2 bindIsClusterIDinList()

Checks for a cluster ID in the cluster ID list of a binding table entry. This function assumes that the passed in entry is valid.

Prototype

```
byte bindIsClusterIDinList( BindingEntry_t *entry, uint16 clusterId );
```

Parameter Details

entry – pointer to a valid binding table entry. The function will search this entry.

clusterId – 16 bit cluster ID to search for.

Return

true if cluster ID is in the list, false if not found.

3.4.2.4 Binding Table Statistics**3.4.2.4.1 bindNumBoundTo()**

Use this function to count all binding table entries with the passed in address and endpoint.

Prototype

```
byte bindNumBoundTo( zAddrType_t *devAddr, byte devEpInt, byte srcMode );
```

Parameter Details

devAddr – address to search the binding table. Works for addrMode equal to either Addr16Bit, Addr64Bit or AddrGroup.

devEpInt – endpoint.

srcMode – true to search source addresses, false to search destination addresses.

Return

Number of binding entries found with he passed in address and endpoint.

3.4.2.4.2 bindNumOfEntries()

This function returns the number of binding entries in the binding table. Each number cluster counts as an entry. So, if there are 2 record entries but each entry has 3 cluster IDs, then the number returned by this function will be 6.

Prototype

```
uint16 bindNumOfEntries( void );
```

Parameter Details

None.

Return

Number of entries found.

3.4.2.4.3 bindCapacity()

This function returns the maximum number of binding entries possible and the number used in the binding table. In this case, it is counting records (not cluster IDs). So, if there are 2 record entries but each entry has 3 cluster IDs, then the number returned in `usedEntries` will be 2.

Prototype

```
void bindCapacity( uint16 *maxEntries, uint16 *usedEntries );
```

Parameter Details

`maxEntries` – pointer to maximum entries variable (output for this function). Number of possible binding table entries. This number can be changed by changing `NWK_MAX_BINDING_ENTRIES` in `f8wConfig.cfg`.

`usedEntries` – pointer to used entries variable (output for this function). Of the possible, this is the number of entries already used.

Return

None.

3.4.2.5 Binding Table Non-Volatile Storage

To use any function in this section, `NV_RESTORE` must be defined in compiler preprocessor section or in `f8wConfig.cfg`.

`BindWriteNV` is the only Binding NV function recommended for user application usage. There are other binding NV functions, but binding NV init and read are called automatically on device startup.

3.4.2.5.1 BindWriteNV()

This function will write the binding table to non-volatile memory, to be called if the user application changes anything in the binding table (add, remove or change). If the binding table is updated normally through ZDO messages, the function will be called by ZDO and doesn't need to be called by the user application.

Prototype

```
void BindWriteNV( void );
```

Parameter Details

None.

Return

None.

3.4.3 Group Table Management

The APS group table is a link list defined with allocated RAM [`osal_mem_alloc()`], so as groups are added to the table the amount of OSAL heap used increases. The maximum table size can be changed by adjusting `APS_MAX_GROUPS` in `f8wConfig.cfg`. The table is defined in `nwk_globals.c`. To use the group table functions include "aps_groups.h" in your application.

3.4.3.1 Group Table Structures

3.4.3.1.1 Group Item - `aps_Group_t`

This structure is the group item and contains the group ID and a text group name.

```
typedef struct
{
    uint16 ID;                // Unique to this table
    uint8  name[APS_GROUP_NAME_LEN]; // Human readable name of group
} aps_Group_t;
```

ID – 16 bit group ID. This is the group ID that is set over the air.

name – text group name (Human readable). APS_GROUP_NAME_LEN is defined as 16 and is not changeable.

3.4.3.1.2 Group Table Entry - `apsGroupItem_t`

This structure is a Group Table record (entry). The group table is a linked list and it is highly recommended that you use the Group Table search and maintenance functions (next sections) to transverse the group table.

```
typedef struct apsGroupItem
{
    struct apsGroupItem *next;
    uint8                endpoint;
    aps_Group_t          group;
} apsGroupItem_t;
```

next – points to the next group table entry. The group table is a linked list. A NULL indicates that this is the last entry in the list. It is highly recommended that you use the Group Table search and maintenance functions (next sections) to transverse the group table.

endpoint – the endpoint that will receive messages sent to the group in the group field.

group – group ID and group name.

3.4.3.2 Group Table Maintenance

3.4.3.2.1 `aps_AddGroup()`

Call this function to add a group into the group table. Define an `aps_Group_t` item, fill it in, then call this function. If NV_RESTORE is enabled, this function will save the update in non-volatile memory.

Prototype

```
ZStatus_t aps_AddGroup( uint8 endpoint, aps_Group_t *group );
```

Parameter Details

endpoint – the endpoint that will receive messages sent to the group in the group field.

group - group ID and group name to add into the group table.

Return

ZSuccess if add was successful. Errors are ZApsDuplicateEntry, ZApsTableFull, or ZMemError (all defined in ZComDef.h).

3.4.3.2.2 aps_RemoveGroup()

Call this function to remove a group from the group table. If NV_RESTORE is enabled, this function will save the update in non-volatile memory.

Prototype

```
uint8 aps_RemoveGroup( uint8 endpoint, uint16 groupID );
```

Parameter Details

endpoint – the endpoint to remove from a group.

groupID - group ID of the group to remove from the table.

Return

true if remove is successful, false if not the group/endpoint is not found.

3.4.3.2.3 aps_RemoveAllGroup ()

Call this function to remove all groups for a given endpoint from the group table. If NV_RESTORE is enabled, this function will save the update in non-volatile memory.

Prototype

```
void aps_RemoveAllGroup( uint8 endpoint );
```

Parameter Details

endpoint – the endpoint to remove from the group table.

Return

None.

3.4.3.3 Group Table Searching**3.4.3.3.1 aps_FindGroup ()**

Call this function to find a group in the group table for an endpoint and group ID.

Prototype

```
aps_Group_t *aps_FindGroup( uint8 endpoint, uint16 groupID );
```

Parameter Details

endpoint – the endpoint that will receive messages sent to the group in the group field.

groupID - group ID.

Return

Pointer to group item found or NULL if not found.

3.4.3.3.2 aps_FindGroupForEndpoint ()

Call this function to find an endpoint from a group ID. This function can be used to skip past an endpoint, then return the next endpoint. Use this function to find all the endpoints for a group ID.

Prototype

```
uint8 aps_FindGroupForEndpoint( uint16 groupID, uint8 lastEP );
```

Parameter Details

groupID - group ID searching for.

lastEP – endpoint to skip past first before returning an endpoint. Use APS_GROUPS_FIND_FIRST to indicate that you want the first endpoint found.

Return

Returns the endpoint that matches the groupID and lastEP criteria, or APS_GROUPS_EP_NOT_FOUND if no (more) endpoints found.

3.4.3.3.3 aps_FindAllGroupsForEndpoint()

Call this function to get a list of all endpoints belong to a group. The caller must provide the space to copy the groups into.

Prototype

```
uint8 aps_FindAllGroupsForEndpoint( uint8 endpoint, uint16 *groupList );
```

Parameter Details

endpoint – endpoint for search.

groupList – pointer to a place to build the list of groups that the endpoint belongs. The caller must provide the memory. To be safe, the caller should create (local or allocated) an array of APS_MAX_GROUPS 16-bit items.

Return

Returns the number of endpoints copied.

3.4.3.3.4 **aps_CountGroups()**

Call this function to get a count of the number of groups that a given endpoint belongs.

Prototype

```
uint8 aps_CountGroups( uint8 endpoint );
```

Parameter Details

endpoint – endpoint for search.

Return

Returns the number of groups.

3.4.3.3.5 **aps_CountAllGroups ()**

Call this function to get the number of entries in the group table.

Prototype

```
uint8 aps_CountAllGroups( void );
```

Parameter Details

None.

Return

Returns the number of groups.

3.4.3.4 **Group Table Non-Volatile Storage**

If NV_RESTORE is defined in either the compiler preprocessor section or in f8wConfig.cfg, the group table will automatically save when a group add or remove is performed. Group table NV init and restore functions are automatically called on device startup.

If a group table entry is modified by the user application, it must call Aps_GroupsWriteNV() directly.

3.4.3.4.1 **aps_GroupsWriteNV()**

This function will write the group table to non-volatile memory, and is to be called if the user application changes anything in a group table entry (other than add, remove or remove all). If the group table is updated normally through group add, remove or remove all functions, there is no need to call this function.

Prototype

```
void aps_GroupsWriteNV( void );
```

Parameter Details

None.

Return

None.

3.4.4 Quick Address Lookup

The APS provides a couple of functions to provide quick address conversion (lookup). These functions allow you to convert from short to IEEE address (or IEEE to short) if the lookup has already been done and stored in the address manager (ref. network layer) or if it's your own address.

3.4.4.1 APSME_LookupExtAddr()

This function will look up the extended (IEEE) address based on a network (short) address if the address is already in the Address Manager. It does NOT start a network (over-the-air) IEEE lookup.

Prototype

```
uint8 APSME_LookupExtAddr(uint16 nwkJddr, uint8* extAddr );
```

Parameter Details

nwkJddr – this is the address you have and would like this function to use to lookup the extended address.

extAddr – this is the address you would like looked up. This is pointer to memory that this function will copy in the IEEE address when found.

Return

true when found, false when not found.

3.4.4.2 APSME_LookupNwkJddr ()

This function will look up the network (short) address based on an extended (IEEE) address if the address is already in the Address Manager. It does NOT start a network (over-the-air) short address lookup.

Prototype

```
uint8 APSME_LookupNwkJddr( uint8* extAddr, uint16* nwkJddr );
```

Parameter Details

nwkJddr – this is the address you would like found. This is pointer to memory that this function will copy in the short address when found.

extAddr – this is the address you and would like this function to use to lookup the extended address. This is pointer to memory that this function will copy in the IEEE address when found.

Return

true when found, false when not found.

3.5 Network Layer (NWK)

The NWK provides the following functionality accessible to the higher layers:

- Network Management
- Address Management
- Network Variables and Utility Functions

Besides the management functions, NWK also provides data services that are not accessible to the application. Applications should send data through the AF data interface [AF_DataRequest ()].

3.5.1 Network Management

3.5.1.1 NLME_NetworkDiscoveryRequest()

This function requests the network layer to discover neighboring routers. This function should be called before joining to perform a network scan. The scan confirm (results) will be returned in ZDO_NetworkDiscoveryConfirmCB() callback. It is best not to use this function (unless you thoroughly understand the network joining process) and instead use ZDO_StartDevice().

Prototype

```
ZStatus_t NLME_NetworkDiscoveryRequest( uint32 ScanChannels,
byte ScanDuration );
```

Parameter Details

ScanChannels – Channels on which to do the discovery. This field is a bit map with each bit representing a channel to scan. Only channels 11 - 26 (0x07FFF800) are available for 2.4 GHz.

ScanDuration – This specified how long each channel should be scanned for other networks before the new network is started. Its range is the same as the beacon order:

Name	Value	Description
BEACON_ORDER_15_MSEC	0	15.36 milliseconds
BEACON_ORDER_30_MSEC	1	30.72 milliseconds
BEACON_ORDER_60_MSEC	2	61.44 milliseconds
BEACON_ORDER_120_MSEC	3	122.88 milliseconds
BEACON_ORDER_240_MSEC	4	245.76 milliseconds
BEACON_ORDER_480_MSEC	5	491.52 milliseconds
BEACON_ORDER_1_SECOND	6	983.04 milliseconds

BEACON_ORDER_2_SECONDS	7	1966.08 milliseconds
BEACON_ORDER_4_SECONDS	8	3932.16 milliseconds
BEACON_ORDER_7_5_SECONDS	9	7864.32 milliseconds
BEACON_ORDER_15_SECONDS	10	15728.64 milliseconds
BEACON_ORDER_31_SECONDS	11	31457.28 milliseconds
BEACON_ORDER_1_MINUTE	12	62914.58 milliseconds
BEACON_ORDER_2_MINUTES	13	125829.12 milliseconds
BEACON_ORDER_4_MINUTES	14	251658.24 milliseconds
BEACON_ORDER_NO_BEACONS	15	No Beacons transmitted

Return

ZStatus_t – status values defined in ZComDef.h.

3.5.1.2 NLME_NwkDiscReq2()

This function requests the network layer to discover neighboring routers. Use this function to perform a network scan but you are NOT expecting to join. The scan confirm (results) will be returned in ZDO_NetworkDiscoveryConfirmCB() callback. After the callback (results), call NLME_NwkDiscTerm() to clean up this action.

Prototype

ZStatus_t NLME_NwkDiscReq2(NLME_ScanFields_t* fields);

Parameter Details

fields – scan structure:

```
typedef struct
{
    uint32 channels;
    uint8 duration;
    uint8 scanType;
    uint8 scanApp;
} NLME_ScanFields_t;
```

channels – Channels on which to do the discovery. This field is a bit map with each bit representing a channel to scan. Only channels 11 - 26 (0x07FFF800) are available for 2.4 GHz.

duration – This specified how long each channel should be scanned for other networks before the new network is started. Its range is the same as the beacon order.

scanType – valid types are: ZMAC_ED_SCAN, ZMAC_ACTIVE_SCAN, ZMAC_PASSIVE_SCAN, MAC_SCAN_PASSIVE, and ZMAC_ORPHAN_SCAN (all defined in zmac_internal.h).

scanApp – valid apps are: NLME_ED_SCAN, NLME_DISC_SCAN, and NLME_PID_SCAN (defined in NLMEDE.h).

Return

ZStatus_t – status values defined in ZComDef.h.

3.5.1.3 NLME_NwkDiscTerm()

This function will clean up the NLME_NwkDiscReq2 () action.

Prototype

```
void NLME_NwkDiscTerm( void );
```

Parameter Details

None.

Return

None.

3.5.1.4 NLME_NetworkFormationRequest()

This function allows the next higher layer to request that the device form a new network. For Zigbee Coordinators this function was be called with parameter DistributedNetwork as FALSE, the function will assign it the short address of 0x0000. Routers must called this function with parameter DistributedNetwork as TRUE, if parameter DistributedNetworkAddress has a valid address (0x0001-0xFFFF7) then the function will assign this address as the device short address, if the provided DistributedNetworkAddress is an invalid value the function will randomly generate a valid short address. The result of this action (status) is returned to the ZDO_NetworkFormationConfirmCB () callback. It is best not to use this function directly and instead use ZDO_StartDevice ().

Prototype

```
ZStatus_t NLME_NetworkFormationRequest( uint16 PanId, uint32 ScanChannels,
                                         byte ScanDuration, byte BeaconOrder,
                                         byte SuperframeOrder, byte BatteryLifeExtension,
                                         bool DistributedNetwork, uint16 DistributedNetworkAddress );
```

Parameter Details

PanId – This field specifies the identifier to be used for the network that will be started by this device. Valid range is from 0 to 0xFFFE. If a value of 0xFFFF is used, the NWK layer will choose the PanId to be used for the network. If a network is found with the same PAN ID here, this PAN ID will be incremented until it is unique (to the scanned PAN IDs).

ScanChannels – Channels on which to do the discovery. This field is a bit map with each bit representing a channel to scan. Only channels 11 - 26 (0x07FFF800) are available for 2.4 GHz.

ScanDuration – This specified how long each channel should be scanned for other networks before the new network is started. Its range is the same as the beacon order.

BeaconOrder – This field should be BEACON_ORDER_NO_BEACONS.

`SuperframeOrder` - This field should be `BEACON_ORDER_NO_BEACONS`.

`BatteryLifeExtension` - If this value is `TRUE`, the NWK layer will request that the ZigBee coordinator is started supporting battery life extension mode (see [R3] for details on this mode). If the value is `FALSE`, the NWK layer will request that the ZigBee coordinator is started without supporting battery life extension mode.

`DistributedNetwork` - If this value is `TRUE`, the function will try to use `DistributedNetworkAddress` as the device short address or randomly generate it.

`DistributedNetworkAddress` - If this value is a valid address (0x0001-0xFFFF7) and `DistributedNetwork` is `TRUE`, this value will be the device's short address. If it is an invalid address and `DistributedNetwork` is `TRUE`, then the function will randomly generate the device's short address. If `DistributedNetwork` is `FALSE`, this parameter is not used.

Return

`ZStatus_t` - status values defined in `ZComDef.h`.

3.5.1.5 NLME_StartRouterRequest()

This function allows the next higher layer to request that the device to start functioning as a router. The result of this action (status) is returned to the `ZDO_StartRouterConfirmCB()` callback. It is best not to use this function directly and instead use `ZDO_StartDevice()`.

Prototype

```
ZStatus_t NLME_StartRouterRequest( byte BeaconOrder, byte SuperframeOrder,
                                   byte BatteryLifeExtension );
```

Parameter Details

`BeaconOrder` -This field should be `BEACON_ORDER_NO_BEACONS`.

`SuperframeOrder` - This field should be `BEACON_ORDER_NO_BEACONS`.

`BatteryLifeExtension` - If this value is `TRUE`, the NWK layer will request that the router is started supporting battery life extension mode (see [R3] for details on this mode). If the value is `FALSE`, the NWK layer will request that the router is started without supporting battery life extension mode.

Return

`ZStatus_t` - status values defined in `ZComDef.h`.

3.5.1.6 NLME_JoinRequest()

This function allows the next higher layer to request that the device to join itself to a network. The result of this action (status) is returned to the `ZDO_JoinConfirmCB()` callback. It is best not to use this function directly and instead use `ZDO_StartDevice()`.

Prototype

```
ZStatus_t NLME_JoinRequest( uint8 *ExtendedPANID, uint16 PanId, byte Channel, byte CapabilityInfo );
```


Parameter Details

`ExtendedPANID` – This field contains the extended PAN ID of the network that you are trying to join.

`PanId` – This field specifies the identifier to be used for the network that this device will join. Valid range is from 0 to 0x3FFF and should be obtained from a scan.

`Channel` – the channel that the wanted network is located. Only channels 11 - 26 are available for 2.4 GHz.

`CapabilityInfo` – Specifies the operational capabilities of the joining device:

Types	Bit
<code>CAPINFO_ALTPANCOORD</code>	0x01
<code>CAPINFO_DEVICETYPE_FFD</code>	0x02
<code>CAPINFO_POWER_AC</code>	0x04
<code>CAPINFO_RCVR_ON_IDLE</code>	0x08
<code>CAPINFO_SECURITY_CAPABLE</code>	0x40
<code>CAPINFO_ALLOC_ADDR</code>	0x80

Return

`ZStatus_t` – status values defined in `ZComDef.h`.

3.5.1.7 NLME_ReJoinRequest()

Use this function to rejoin a network that this device has already been joined to. The result of this action (status) is returned to the `ZDO_JoinConfirmCB()` callback.

Prototype

```
ZStatus_t NLME_ReJoinRequest(uint8 *ExtendedPANID, uint8 channel );
```

Parameter Details

`ExtendedPANID` – This field contains the extended PAN ID of the network that you are trying to join.

`Channel` – the channel that the wanted network is located. Only channels 11 - 26 are available for 2.4 GHz.

Return

`ZStatus_t` – status values defined in `ZComDef.h`.

3.5.1.8 NLME_OrphanJoinRequest()

This function requests the network layer to orphan join into the network. This function is a scan with an implied join. The result of this action (status) is returned to the `ZDO_JoinConfirmCB()` callback. It is best not to use this function (unless you thoroughly understand the network joining process) and instead use `ZDO_StartDevice()`.

Prototype

```
ZStatus_t NLME_OrphanJoinRequest( uint32 ScanChannels, byte ScanDuration );
```

Parameter Details

ScanChannels – Channels on which to do the discovery. This field is a bit map with each bit representing a channel to scan. Only channels 11 - 26 (0x07FFF800) are available for 2.4 GHz.

ScanDuration – This specified how long each channel should be scanned for other networks before the new network is started. Its range is the same as the beacon order.

Return

ZStatus_t – status values defined in `ZComDef.h`.

3.5.1.9 NLME_PermitJoiningRequest()

This function defines how the next higher layer of a coordinator or router device may permit devices to join its network for a fixed period.

Prototype

```
ZStatus_t NLME_PermitJoiningRequest( byte PermitDuration );
```

Parameter Details

PermitDuration – The length of time during which the a device (coordinator or router) will be allowing associations. The values 0x00 and 0xff indicate that permission disabled or enabled, respectively without a time limit. Values 0x01 – 0xFE are the number of seconds to allow joining.

Return

ZStatus_t – status values defined in `ZComDef.h`.

3.5.1.10 NLME_DirectJoinRequest()

This function allows the next higher layer to request the NWK layer on a router or coordinator device to add another device as its child device.

Prototype

```
ZStatus_t NLME_DirectJoinRequest( byte *DevExtAddress, byte capInfo );
```

Parameter Details

DevExtAddress – The pointer to the IEEE address of the device that should be added as a child of this device .

capInfo – Specifies the operational capabilities of the joining device. See the `CapabilityInfo` in the `NLME_JoinRequest()`.

Return

ZStatus_t - status values defined in ZComDef.h.

3.5.1.11 NLME_LeaveReq()

This function allows the next higher layer to request that it or another device leaves the network. In a Non-Pro network, calling this function will NOT cause the parent of the leaving device to re-allocate the device's address.

Prototype

```
ZStatus_t NLME_LeaveReq( NLME_LeaveReq_t* req );
```

Parameter Details

req - Leave Request structure:

```
typedef struct
{
    uint8* extAddr;
    uint8  removeChildren;
    uint8  rejoin;
    uint8  silent;
} NLME_LeaveReq_t;
```

extAddr - Extended address of the leaving device.

removeChildren - true for the devices children to also leave. false for just the device to leave. ONLY false should be use.

rejoin - true will allow the device to rejoin the network. false if the device is not allowed to rejoin the network.

silent - true will. false if the.

Return

ZStatus_t - status values defined in ZComDef.h.

3.5.1.12 NLME_RemoveChild()

This function checks if the specified EXT address is a child's address, if so it is removed. If the "dealloc" argument is true then the NWK address for the child is freed up for reuse. This function will only deallocate end device children addresses and the deallocate address is only useful in a non-Pro tree network.

Prototype

```
void NLME_RemoveChild( uint8* extAddr, uint8 dealloc );
```

Parameter Details

extAddr - Extended address of the device to remove.

dealloc - true will. false if the.

Return

ZStatus_t - status values defined in ZComDef.h.

3.5.1.13 NwkPollReq()

Call this function to manually request a MAC data request. This function is for end devices only. Normally, for end devices, the polling is automatically handled by the network layer and the application can manipulate the poll rate by calling NLME_SetPollRate(). If the poll rate is set to 0, the application can manually do the polls by call this function.

Prototype

```
ZMacStatus_t NwkPollReq( byte securityEnable );
```

Parameter Details

securityEnable - Always set this field to false.

Return

ZMacStatus_t is the same as ZStatus_t - status values defined in ZComDef.h.

3.5.1.14 NLME_SetPollRate()

Use this function to set/change the Network Poll Rate. This function is for end devices only. Normally, for end devices, the polling is automatically handled by the network layer and the application can manipulate the poll rate by calling NLME_SetPollRate().

Prototype

```
void NLME_SetPollRate( uint16 newRate );
```

Parameter Details

newRate - the number of milliseconds between data polls to its parent. A value of 0 disables polling and a value of 1 does a onetime data poll without setting the poll rate.

Return

None.

3.5.1.15 NLME_SetQueuedPollRate()

Use this function to set/change the Queued Poll Rate. This function is for end devices only. If a poll for data results in a data message, the poll rate is immediately set to the Queued Poll Rate to drain the parent of queued data.

Prototype

```
void NLME_SetQueuedPollRate( uint16 newRate );
```

Parameter Details

`newRate` – the number of milliseconds for the queued poll rate. A value of 0 disables the queue poll and 0x01 – 0xFF represent the number of milliseconds.

Return

None.

3.5.1.16 NLME_SetResponseRate()

Use this function to set/change the Response Poll Rate. This function is for end devices only. This is the poll rate after sending a data request, the idea being that we are expecting a response quickly (instead of wait the normal poll rate).

Prototype

```
void NLME_SetResponseRate( uint16 newRate );
```

Parameter Details

`newRate` – the number of milliseconds for the response poll. A value of 0 disables the response poll and 0x01 – 0xFF represent the number of milliseconds.

Return

None.

3.5.1.17 NLME_RouteDiscoveryRequest()

Use this function to send route request to issue route discovery. The route request could be either a unicast route request which establishes routing path from the local device to another device, or a many-to-one route request which establishes routes from all other devices in the network to the concentrator device (which is supposed to be the local device that issues many-to-one route request).

Prototype

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

Parameter Details

`DstAddress` – the target address of the route request. In the case of unicast route request, this has to be the address of the destination device that the local device is trying to establish route to. In the case of many-to-one route request, the parameter will be overwritten in the function call with many-to-one destination address which is typically 0xFFFC which means all routers and coordinator.

`options` – route discovery option bitmask. Values are listed in the following table.

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

Radius – broadcast radius of the route request packet. If it is set to zero, then the radius will be set to the default broadcast radius.

Return

ZStatus_t – status values defined in `ZComDef.h`.

3.5.2 Address Management

The Address Manager provides low level address management. Currently, the user should not access this module directly. Support for local address lookup is provided by the functions `APSME_LookupExtAddr` and `APSME_LookupNwkAddr`. Support for remote address lookup is provided by the functions `ZDP_IIEEEAddrReq` and `ZDP_NwkAddrReq`. If the user requires remote address storage on the local device, a proprietary solution should be implemented.

3.5.2.1 Network Variables and Utility Functions

3.5.2.1.1 NLME_GetExtAddr()

This function will return a pointer to the device's IEEE 64 bit address.

Prototype

```
byte *NLME_GetExtAddr( void );
```

Parameter Details

None.

Return

Pointer to the 64-bit extended address.

3.5.2.1.2 NLME_GetShortAddr()

This function will return the device's network (short - 16 bit) address.

Prototype

```
uint16 NLME_GetShortAddr( void );
```

Parameter Details

None.

Return

16-bit network (short) address.

3.5.2.1.3 NLME_GetCoordShortAddr()

This function will return the device's parent's network (short - 16 bit) address. This is NOT the Zigbee Coordinator's short address (it's always 0x0000). In MAC terms, the parent is called a coordinator.

Prototype

```
uint16 NLME_GetCoordShortAddr( void );
```

Parameter Details

none.

Return

16-bit network (short) address.

3.5.2.1.4 NLME_GetCoordExtAddr()

This function will get the device's parent's IEEE (64 bit) address. This is NOT the Zigbee Coordinator's extended address. In MAC terms, the parent is called a coordinator.

Prototype

```
void NLME_GetCoordExtAddr( byte *buf );
```

Parameter Details

`buf` – Pointer to buffer for the parent's extended address.

Return

None.

3.5.2.1.5 NLME_SetRequest()

This function allows the next higher layer to set the value of a NIB (network information base) attribute.

Prototype

```
ZStatus_t NLME_SetRequest( ZNwkAttributes_t NIBAttribute,  
                           uint16 Index,  
                           void *Value );
```

Parameter Details

NIBAttribute - Only supported attributes are: `nwkProtocolVersion`

Index - not used.

Value - Pointer to a memory location that contains the value of the attribute.

Return

ZStatus_t - status values defined in `ZComDef.h`.

3.5.2.1.6 NLME_GetRequest()

This function allows the next higher layer to get the value of a NIB (network information base) attribute.

Prototype

```
ZStatus_t NLME_GetRequest( ZNwkAttributes_t NIBAttribute,  
                           uint16 Index, void *Value );
```

Parameter Details

NIBAttribute - Only supported attributes are:

- `nwkCapabilityInfo`
- `nwkNumNeighborTableEntries`
- `nwkNeighborTable`
- `nwkNumRoutingTableEntries`
- `nwkRoutingTable`

Index - used to index into tables.

Value - Pointer to a memory location that contains the value of the attribute.

Return

ZStatus_t - status values defined in `ZComDef.h`.

3.5.2.1.7 NLME_IsAddressBroadcast()

Based on device capabilities this function evaluates the supplied address and determines whether it is a valid broadcast address for this device given its capabilities.

Prototype


```
addr_filter_t NLME_IsAddressBroadcast(uint16 shortAddress);
```

Parameter Details

`shortAddress` - address to be tested.

Return

`addr_filter_t` - result of function has the following types:

Types	Description
ADDR_NOT_BCAST	not a broadcast address of any kind (including reserved values)
ADDR_BCAST_FOR_ME	the address is a broadcast address and I'm the right device type
ADDR_BCAST_NOT_ME	the address is broadcast address but I'm not the target

3.5.2.1.8 NLME_GetProtocolVersion()

This function uses the GET API access to the NIB to retrieve the current protocol version. It is simply a convenience. Otherwise it would be a two-step process (as it is here).

Prototype

```
byte NLME_GetProtocolVersion();
```

Parameter Details

None.

Return

Protocol version in LSB of byte returned. Values are:

Types	Value
ZB_PROT_V1_0	1
ZB_PROT_V1_1	2

3.5.2.1.9 NLME_SetBroadcastFilter()

This function sets a bit mask based on the capabilities of the device. It will be used to process valid broadcast addresses.

Prototype

```
void NLME_SetBroadcastFilter(byte capabilities);
```

Parameter Details

`capabilities` – the device capabilities used to determine what type of broadcast messages the device can handle. This is the same capabilities used to join.

Return

None.

3.5.3 Network Non-Volatile Storage

If `NV_RESTORE` is defined in either the compiler preprocessor section or in `f8wConfig.cfg`, the Network Information Base (NIB) will automatically save when a device joins. NIB NV init and restore functions are automatically called on device startup.

If the NIB (`_NIB`) is modified by the user application, it must call `NLME_UpdateNV()` directly.

3.5.3.1 NLME_UpdateNV()

This function will write the NIB to non-volatile memory, and is to be called if the user application changes anything in the NIB. If the NIB is updated normally through joining, there is no need to call this function.

Prototype

```
void NLME_UpdateNV( byte enables );
```

Parameter Details

`enables` – bit mask of items to save:

Name	Value	Description
<code>NWK_NV_NIB_ENABLE</code>	<code>0x01</code>	Save Network Layer NIB.
<code>NWK_NV_DEVICELIST_ENABLE</code>	<code>0x02</code>	Save the Device List
<code>NWK_NV_BINDING_ENABLE</code>	<code>0x04</code>	Save the Binding Table
<code>NWK_NV_ADDRMGR_ENABLE</code>	<code>0x08</code>	Save the Address Manager Table

Return

None.

3.5.4 PAN ID Conflict

Since the 16-bit PAN ID is not a unique number, there is a possibility of a PAN ID conflict in the local neighborhood. A node that has detected a PAN ID conflict sends a Network Report Command to the designated Network Manager. On receipt of the Network Report command, the Network Manager selects a new PAN ID for the network. Once a new PAN ID has been selected, the Network Manager broadcasts a Network Update command (conveying the new PAN ID) to the network. On receipt of a Network Update command, all the devices change their current PAN ID to new selected PAN ID.

3.5.4.1 NLME_SendNetworkReport ()

This function sends out a Network Report command. The Network Report command allows a device to report network events to the device identified by the address contained in the `nwkManagerAddr` in the NIB. Such events are radio channel condition and PAN ID conflicts.

Prototype

```
void NLME_SendNetworkReport ( uint16 dstAddr, uint8 reportType, uint8 *EPID, uint8 reportInfoCnt,  
                             uint16 *panIDs );
```

Parameter Details

`dstAddr` – The destination address of the message.

`reportType` – The report command identifier.

`EPID` – The 64-bit EPID that identifies the network that the reporting device is a member of.

`reportInfoCnt` – The number of the records contained in the `panIDs` field.

`panIDs` – The list of 16-bit PAN IDs that are operating in the neighborhood of the reporting device.

Return

None.

3.5.4.2 NLME_SendNetworkUpdate()

This function sends out a Network Update command. The Network Update command allows the device identified by the `nwkManagerAddr` attribute of the NIB to broadcast the change of configuration information to all devices in the network. For example, broadcasting the fact that the network is about to change its short PAN identifier.

Prototype

```
void NLME_SendNetworkUpdate ( uint16 dstAddr, uint8 updateType, uint8 *EPID,  
                             uint8 updateId, uint16 newPID );
```

Parameter Details

`dstAddr` – The destination address of the message.

`updateType` – The update command identifier.

`EPID` – The 64-bit EPID that identifies the network that is to be updated..

`updateId` – The current value of the `nwkUpdateId` attribute of the device sending the command.

`newPID` – The new PAN ID for the network to use.

Return

None.

3.5.5 Inter-PAN Transmission

The Stub APS layer provides interfaces to exchange Inter-PAN data, switch channel for Inter-PAN communication and check for Inter-PAN messages.

3.5.5.1 StubAPS_SetInterPanChannel()

This function allows the application to switch channel for inter-PAN communication.

Prototype

```
ZStatus_t StubAPS_SetInterPanChannel( uint8 channel );
```

Parameter Details

`channel` – new channel.

Return

`ZStatus_t` – status values defined in `ZComDef.h`.

3.5.5.2 StubAPS_SetIntraPanChannel()

This function allows the application to switch channel back to the NIB channel.

Prototype

```
ZStatus_t StubAPS_SetIntraPanChannel( void );
```

Parameter Details

None.

Return

`ZStatus_t` – status values defined in `ZComDef.h`.

3.5.5.3 StubAPS_InterPan()

This function allows the application to see if it's communicating to a different PAN.

Prototype

```
uint8 StubAPS_InterPan( uint16 panId, uint8 endPoint );
```

Parameter Details

`panId` – PAN ID.
`endPoint` – Endpoint.

Return

uint8 – TRUE if PAN is Inter-PAN, FALSE otherwise.

3.5.5.4 StubAPS_RegisterApp()

This function allows the application to register itself with the Stub APS layer.

Prototype

```
void StubAPS_RegisterApp( endPointDesc_t *epDesc );
```

Parameter Details

epDesc – application's endpoint descriptor.

Return

None.

3.6 Green Power Layer

The Green Power layer provides interface to send and receive data from the Green Power Stubs. It also provides callback notifications on some commissioning process.

3.6.1 GP_DataInd()

This primitive is generated and passed to the application in the event of the receipt, by the dGP stub, of a MCPS-DATA.indication primitive from the MAC sub-layer, containing a frame that was generated by the GPD, and that was intended for the receiving device.

Prototype

```
uint8 GP_DataInd (gp_DataInd_t *gp_DataInd);
```

Parameter Details

gp_DataInd – pointer to the structure containing the GPDPF.

Return

bool – TRUE if the frame will be released, FALSE if the frame buffer will be used by the higher layer.

3.6.2 GP_DataReq()

Primitive by GP EndPoint to pass to dGP stub a request to send GPDPF to a GPD.

Prototype

```
bool GP_DataReq(gp_DataReq_t *gp_DataReq) ;
```

Parameter Details

gp_DataReq – pointer to the structure containing the GP data request to be send.

Return

bool – TRUE if the frame will be released, FALSE if the frame buffer will be used by the receiving layer.

3.6.3 GP_DataCnf()

This primitive is generated by the lower layers and passed to the Green Power EndPoint after the GP-DATA.request has been handled.

Prototype

```
void GP_DataCnf(gp_DataCnf_t *gp_DataCnf);
```

Parameter Details

gp_DataCnf – pointer to the structure containing the status of the Data request process and the handle that identifies that request.

Return

None.

3.6.4 GP_SecReq()

This primitive is generated by the dGP stub and passed to the Green Power EndPoint on reception of GPDF. The application endpoint must validate duplicate frames and if encrypted it must get the right key to be used.

Prototype

```
uint8 GP_SecReq(gp_SecReq_t *gp_SecReq);
```

Parameter Details

gp_SecReq – pointer to the structure containing the GPDF to be validated.

Return

bool – TRUE if the frame will be released, FALSE if the frame buffer will be used by the receiving layer.

3.6.5 GP_SecRsp()

This primitive is generated by the Green Power EndPoint and passed to the dGP stub on reception of GP-SEC.request.

Prototype

```
void GP_SecRsp(gp_SecRsp_t *gp_SecRsp);
```

Parameter Details

`gp_SecReq` –pointer to the structure containing the response to the `GP_SecReq`. In the structure status is indicated the action to be taken with request and the key to be used to decrypt the frame if protected.

Return

None.

3.7 ZMac Layer (ZMAC)

The ZMAC layer provides packet translation functionality between the 802.15.4 MAC and the ZigBee NWK layers.

3.7.1 ZMacSetTransmitPower()

This function allows the application to request the MAC to set the transmit power level.

Prototype

```
uint8 ZMacSetTransmitPower( ZMacTransmitPower_t level );
```

Parameter Details

`level` – Valid power level setting as defined in `ZMAC.h`.

Return

`ZMacSuccess` – if the MAC PIB attribute `MAC_PHY_TRANSMIT_POWER_SIGNED` is found.

`ZMacUnsupportedAttribute` – if `MAC_PHY_TRANSMIT_POWER_SIGNED` is not found.

3.7.2 ZMacLqiAdjustMode()

This function allows the application to set the `ZMacLqiAdjust` mode of operation. The enumerated list of LQI Adjust modes is located in `ZMAC.h` with the default mode setting of `LQI_ADJ_OFF`.

Prototype

```
ZMacLqiAdjust_t ZMacLqiAdjustMode( ZMacLqiAdjust_t mode );
```

Parameter Details

`mode` – new LQI Adjust mode of operation:

Mode	Value	Description
<code>LQI_ADJ_OFF</code>	<code>0x00</code>	Disable LQI Adjust feature
<code>LQI_ADJ_MODE1</code>	<code>0x01</code>	Enable LQI Adjust mode 1

LQI_ADJ_MODE2	0x02	Enable LQI Adjust mode 2
LQI_ADJ_GET	0xFF	Return current LQI Adjust mode

Return

zMacLqiAdjust_t - current mode of LQI Adjust feature:

Type	Value	Description
LQI_ADJ_OFF	0x00	LQI Adjust feature is disabled
LQI_ADJ_MODE1	0x01	LQI Adjust mode 1 is enabled
LQI_ADJ_MODE2	0x02	LQI Adjust mode 2 is enabled