Chipcon Products from Texas Instruments

Z-Stack Application Programming Interface

Document Number: F8W-2006-0021

Texas Instruments, Inc.

San Diego, California USA

(619) 542-1200

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	08/7/2007

Table of Contents

1. Introduction

1.1 Purpose

This document provides the Application Programmers Interface (API) for the Z-Stack components provided as part of the ZigBee-2006 compliant Z-Stack 1.4.0 Release.

1.2 Scope

This document enumerates the implemented API of all components provided as part of the ZigBee-2006 compliant 1.4.0 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
MSG	Message
NHLE	Next Higher Layer Entity
NWK	Network
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 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.

2.2 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.

2.3 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.

2.4 NWK

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

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 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.

3.1.1 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.

3.1.2 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. For the application to take control of the device network start, it must include HOLD_AUTO_START as a compile option and it is recommended that it also include the NV_RESTORE compile option (to save the Zigbee Network State in NV). If the device includes these compile flags it will need to call ZDOInitDevice() to start the device in the network.

3.1.2.1 ZDOInitDevice()

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. To call this function the device MUST have the compile flag HOLD_AUTO_START compiled in.

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);`].

Prototype

```
uint8 ZDOInitDevice( uint16 startDelay );
```

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))
```

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.

3.1.3 ZDO Message Callbacks

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

3.1.3.1 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 (OTA), 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 OTA 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.

3.1.3.2 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.

3.1.4 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_EndDeviceAnnce()	Device_annce
ZDP_ServerDiscReq()	System_Server_Discovery_req
ZDP_ServerDiscRsp()	System_Server_Discovery_rsp

3.1.4.1 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:

ReqType

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.

3.1.4.2 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.

3.1.4.3 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
ZDP_IEEEADDR_REQTYPE_SINGLE	Return only the device's short and extended address
ZDP_IEEEADDR_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.

`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.

3.1.4.4 ZDP_IEEEAddrRsp()

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

3.1.4.5 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 t he 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.

3.1.4.6 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.

3.1.4.7 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.

3.1.4.8 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	
Meaning	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.

3.1.4.9 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.

`DstAddr` - The destination address.

`Status` – The following results:

Status	
Meaning	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.

3.1.4.11 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.

3.1.4.12 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.

3.1.4.13 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.

3.1.4.14 ZDP_MatchDescReq()

This call will build and send an Match Descripton 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.

3.1.4.15 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.

3.1.4.16 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 to16 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.

3.1.4.17 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.

3.1.4.18 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.

3.1.4.19 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.

3.1.4.20 ZDP_EndDeviceAnnce()

This function builds and sends an End_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 end device new 16-bit NWK address and 64-bit IEEE address, as well as the capability of the ZigBee end device. It is sent out as broadcast message.

On receipt of the End_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 End_Device_annce.

Prototype

```
afStatus_t ZDP_EndDeviceAnnce( 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.

3.1.4.21 ZDP_ServerDiscReq()

The function builds and sends a System_Server_Discovery_req request message which contains a 16 bit server mask. The purpose o f 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.

3.1.4.22 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.

3.1.5 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

3.1.5.1 ZDP_EndDeviceBindReq()

This call will build and send an End Device Bind Request (“Hand Binding”). Send this message to attempt a hand bind for this device. After hand binding your 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 epIntf,
      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.

`epIntf` – 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.

3.1.5.2 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.

3.1.5.3 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 SrcEPIntf, byte ClusterID, byte *DestinationAddr, byte DstEPIntf,
      byte SecuritySuite );
```

Parameter Details

`DstAddr` - The destination address.

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

`SrcEPIntf` – endpoint/interface 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.

`DstEPIntf` – endpoint/interface 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.

3.1.5.4 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.

3.1.5.5 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 SrcEPIntf,
                            byte ClusterID,
                            byte *DestinationAddr, byte DstEPIntf,
                            byte SecuritySuite );
```

Parameter Details

`DstAddr` - The destination address.

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

`SrcEPIntf` – endpoint/interface 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.

`DstEPIntf` – endpoint/interface 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.

3.1.5.6 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.

3.1.6 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

3.1.6.1 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.

3.1.6.2 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.

3.1.6.3 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.

3.1.6.4 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.

3.1.6.5 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.

3.1.6.6 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.

3.1.6.7 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.

3.1.6.8 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.

3.1.6.9 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.

3.1.6.10 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.

3.1.6.11 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.

3.1.6.12 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.

3.1.6.13 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.

3.1.6.14 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.

3.1.7 ZDO Parsing Functions

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

3.1.7.1 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()].

3.1.7.2 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.`

3.1.7.3 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.`

3.1.7.4 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.`

3.1.7.5 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()].

3.1.7.6 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.

3.1.7.7 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()].

3.1.7.8 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()].

3.1.7.9 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()].

3.1.7.10 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()].

3.1.7.11 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.

3.1.7.12 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.

3.1.7.13 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.

3.1.7.14 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()].

3.1.7.15 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.
```

3.1.7.16 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.
```

3.1.7.17 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.
```

3.1.7.18 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.

3.1.7.19 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.`

3.1.7.20 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 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.

3.2.1 Overview

The AF provides the following functionality to applications:

- Endpoint Management
- Sending and Receiving Data

3.2.1.1 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.

3.2.1.1.1 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 it 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.

3.2.1.1.2 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
```

3.2.1.1.3 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

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.

3.2.1.1.4 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.

3.2.1.1.5 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.

3.2.1.1.6 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()].

3.2.1.1.7 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.

3.2.1.1.8 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.

3.2.1.1.9 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).

3.2.1.1.10 afEndPoints ()

This function will return an array of endpoints registered. It fills in the passed in 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.

3.2.1.2 Sending Data

3.2.1.2.1 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_FRAGMENTED	0x01	Not to be used.
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_DISCV_ROUTE	0x20	Should always be included.
AF_EN_SECURITY	0x40	Not needed.
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.

`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.

3.2.1.2.2 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 Application Support Sub-Layer (APS)

3.3.1 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.

3.3.2 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.

3.3.2.1 Binding Record Structure – BindingEntry_t

```
typedef struct
{
  uint16 srcIdx;         // Address Manager index
  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;
```

`srcIdx` – Address manager index for the source address. The address manager keeps the IEEE and short address of the source address. The source address here is the source address in the binding record.

`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].

3.3.2.2 Binding Table Maintenance

3.3.2.2.1 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( zAddrType_t *srcAddr, byte srcEpInt,
                                  zAddrType_t *dstAddr, byte dstEpInt,
                                  byte numClusterIds, uint16 *clusterIds );
```

Parameter Details

`srcAddr` – binding record source address. This address type must contain either `Addr16Bit` or `Addr64Bit` address mode with the appropriate `addr` field filled in.

`srcEpInt` – 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.

`dstEpInt` – 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.

3.3.2.2.2 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.

3.3.2.2.3 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.

3.3.2.2.4 bindAddClusterIdToList()

This function will add a cluster ID to the cluster ID list of an existing binding table entry. 41( )-15878252(s)3.457rrr1. /R11 9.9.01205

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.

3.3.2.2.7 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.

3.3.2.3 Binding Table Searching

3.3.2.3.1 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( zAddrType_t *srcAddr, byte srcEpInt,
                                        zAddrType_t *dstAddr, byte dstEpInt );
```

Parameter Details

`srcAddr` – binding record source address. This address type must contain either `Addr16Bit` or `Addr64Bit` address mode with the appropriate `addr` field filled in.

`srcEpInt` – 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.

`dstEpInt` – 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.

3.3.2.3.2 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.

3.3.2.4 Binding Table Statistics

3.3.2.4.1 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.

3.3.2.4.2 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.

3.3.2.4.3 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.

3.3.2.5 Binding Table Non-Volitile 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.

3.3.2.5.1 BindWriteNV()

This function will write the binding table to non-volitale 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.

3.3.3 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.

3.3.3.1 Group Table Structures

3.3.3.1.1 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.

3.3.3.1.2 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.

3.3.3.2 Group Table Maintenance

3.3.3.2.1 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).

3.3.3.2.2 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 removed 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.

3.3.3.2.3 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 removed from the group table.

Return.

none.

3.3.3.3 Group Table Searching

3.3.3.3.1 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.

3.3.3.3.2 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.

3.3.3.3.3 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.

3.3.3.3.4 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.

3.3.3.3.5 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.

3.3.3.4 Group Table Non-Volitile Storage

If `NV_RESTORE` is defined in either the compiler preprocessor section or in f8wConfig.cfg, the group table will automatically save when 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.

3.3.3.4.1 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.

3.3.4 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.

3.3.4.1 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 nwkAddr, uint8* extAddr );
```

Parameter Details

`nwkAddr` – 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.

3.3.4.2 APSME_ LookupNwkAddr ()

This function will look up the network (short) address based on a 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_LookupNwkAddr( uint8* extAddr, uint16* nwkAddr );
```

Parameter Details

`nwkAddr` – 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.

3.4 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 trk( )-3.1(e)8942026.0241(,)(p)-6.0241(

3.4.1.1.1.1.1 Name	3.4.1.1.1.1.2 Value	3.4.1.1.1.1.3 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

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.

Return.

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

3.4.1.1.3 NLME_NwkDiscTerm()

This function will clean up the `NLME_NwkDiscReq2()` action.

Prototype

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

Parameter Details

none.

Return.

none.

3.4.1.1.4 NLME_NetworkFormationRequest()

This function allows the next higher layer to request that the device form a new network and become the Zigbee Coordinator for that network. 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  );
```

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 0x3FFF. 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` – For Zigbee 2006 this field should be `BEACON_ORDER_NO_BEACONS`.

SuperframeOrder – For Zigbee 2006 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.

Return.

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

3.4.1.1.5 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` – For Zigbee 2006 this field should be `BEACON_ORDER_NO_BEACONS`.

SuperframeOrder – For Zigbee 2006 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.

3.4.1.1.6 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:

3.4.1.1.6.1.1 Types	3.4.1.1.6.1.2 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

Return.

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

3.4.1.1.7 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( void );
```

Parameter Details

none.

Return.

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

3.4.1.1.8 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.

3.4.1.1.9 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.

3.4.1.1.10 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.

3.4.1.1.11 NLME_LeaveReq()

This function allows the next higher layer to request that it or another device leaves the network. Currently, 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.

3.4.1.1.12 NLME_RemoveChild()

This function allows the next higher layer to request that

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.

3.4.1.1.13 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.

3.4.1.1.14 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 one time data poll without setting the poll rate.

Return.

none.

3.4.1.1.15 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.

3.4.1.1.16 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.

3.4.1.2 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_IEEEAddrReq and ZDP_NwkAddrReq. If the user requires remote address storage on the local device, a proprietary solution should be implemented.

3.4.1.3 Network Variables and Utility Functions

3.4.1.3.1 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.

3.4.1.3.2 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.

3.4.1.3.3 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.

3.4.1.3.4 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.

3.4.1.3.5 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.

3.4.1.3.6 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.

3.4.1.3.7 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:

3.4.1.3.7.1.1 Types	3.4.1.3.7.1.2 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

3.4.1.3.8 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:

3.4.1.3.8.1.1 Types	3.4.1.3.8.1.2 Value
ZB_PROT_V1_0	1
ZB_PROT_V1_1	2

3.4.1.3.9 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.

3.4.1.4 Network Non-Volitile 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.

3.4.1.4.1 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
NWK_NV_NIB_ENABLE	0x01	Save Network Layer NIB.
NWK_NV_DEVICELIST_ENABLE	0x02	Save the Device List
NWK_NV_BINDING_ENABLE	0x04	Save the Binding Table
NWK_NV_ADDRMGR_ENABLE	0x08	Save the Address Manager Table

Return.

none.