ZigBee 3.0 Stack User Guide

JN-UG-3130
Revision 2.0
18 November 2019

Contents

Part II: Reference Information

Part IV: Appendices

Preface

This manual provides a single point of reference for information relating to the ZigBee 3.0 wireless networking protocol and its associated stack, when implemented on the NXP K32W041, K32W061, and JN518x family of wireless microcontrollers. The manual provides both conceptual and practical information concerning the ZigBee 3.0 protocol and the supporting NXP software. Guidance is provided on the use of the NXP Application Programming Interfaces (APIs) for ZigBee 3.0. The API resources (such as functions, network parameters, enumerations, data types and events) are fully detailed and the manual should be used as a reference resource during ZigBee 3.0 application development.

Note 1: The ZigBee 3.0 protocol employs the ZigBee PRO stack - in particular, Revision 22/ZigBee2017 of this stack. Therefore, this User Guide relates to this stack revision.

Note 2: This User Guide is concerned with the development of applications that operate over the ZigBee PRO stack. These applications may conform to ZigBee 3.0 or may use ZigBee or manufacturer-specific application profiles. ZigBee 3.0 applications are based on ZigBee device types and, in this case, reference must also be made to the *ZigBee 3.0 Devices User Guide (JN-UG-3131)*.

For more detailed information on the ZigBee 3.0 standard, refer to the protocol specifications available from the ZigBee Alliance.

Organisation

This manual is divided into four parts:

- Part I: Concept and Operational Information comprises five chapters:
  - Chapter 1 introduces the ZigBee 3.0 wireless network protocol.
  - Chapter 2 describes the architecture and features of ZigBee 3.0.
  - Chapter 3 introduces the NXP ZigBee PRO stack software.
  - Chapter 4 provides an overview of the ZigBee 3.0 application development environment and process.
  - Chapter 5 describes how to perform common wireless network operations using the functions of the NXP ZigBee 3.0 APIs.

- Part II: Reference Information comprises six chapters:
  - Chapter 6 details the functions and associated resources of the ZigBee Device Objects (ZDO) API.
  - Chapter 7 details the functions and associated resources of the Application Framework (AF) API.
  - Chapter 8 details the functions and associated resources of the ZigBee Device Profile (ZDP) API.
  - Chapter 9 details the general functions and associated resources provided with the NXP ZigBee PRO stack.
  - Chapter 10 details the stack events and the return/status codes used by the ZigBee PRO APIs.
  - Chapter 11 details the ZigBee network parameters.
- Part III: Network Configuration comprises one chapter:
  - Chapter 12 describes how to use the ZPS Configuration Editor.
- Part IV: Appendices contains five appendices that provide various ancillary information, including a description of the handling of ZigBee PRO stack events, a set of application design notes, a description of frame counters, a description of application storage in Flash memory and a glossary of terms used in ZigBee 3.0 networks.

Conventions

Files, folders, functions and parameter types are represented in **bold** type.

Function parameters are represented in *italics* type.

Code fragments are represented in the `Courier New` typeface.

This is a **Tip**. It indicates useful or practical information.

This is a **Note**. It highlights important additional information.

*This is a **Caution**. It warns of situations that may result in equipment malfunction or damage.*

Acronyms and Abbreviations

AF	Application Framework
AIB	APS Information Base
APDU	Application Protocol Data Unit
API	Application Programming Interface
APS	Application Support (sub-layer)
APSDE	Application Support (sub-layer) Data Entity
APSME	Application Support (sub-layer) Management Entity
BDB	Base Device Behaviour
DIO	Digital Input/Output
EPID	Extended PAN ID
HA	Home Automation
HVAC	Heating, Ventilation and Air-Conditioning
IO	Input/Output
ISR	Interrupt Service Routine
MAC	Media Access Control
PAN	Personal Area Network
NIB	NWK Information Base
NPDU	Network Protocol Data Unit
NVM	Non-Volatile Memory Manager
NWK	Network
OS	Operating System
PDU	Protocol Data Unit
PDUM	Protocol Data Unit Manager
PIC	Programmable Interrupt Controller
RF	Radio Frequency
SAP	Service Access Point
SDK	Software Developer’s Kit
UART	Universal Asynchronous Receiver-Transmitter
ZCL	ZigBee Cluster Library
ZCP	ZigBee Compliant Platform
ZDO	ZigBee Device Objects
ZDP	ZigBee Device Profile

ZLL	ZigBee Light Link
ZPS	ZigBee PRO Stack

Related Documents

JN-UG-3131	ZigBee 3.0 Devices User Guide
JN-UG-3132	ZigBee Cluster Library (for ZigBee 3.0) User Guide
JN-UG-3134	ZigBee Green Power (for ZigBee 3.0) User Guide
JN-UG-3133	DK6 Core Utilities User Guide

Support Resources

To access online support resources such as SDKs, Application Notes and User Guides, visit the Wireless Connectivity area of the NXP web site:

www.nxp.com/products/wireless-connectivity

ZigBee resources can be accessed from the ZigBee page, which can be reached via the short-cut **www.nxp.com/zigbee**.

All NXP resources referred to in this manual can be found at the above addresses, unless otherwise stated.

Trademarks

All trademarks are the property of their respective owners.

Chip Compatibility

The software described in this manual can be used on the NXP K32W041, K32W061 and JN518x family of wireless microcontrollers.

Part I: Concept and Operational Information

1. ZigBee Overview

The ZigBee protocol was developed to provide low-power, wireless connectivity for a wide range of network applications concerned with monitoring and control. ZigBee is a worldwide open standard controlled by the ZigBee Alliance. ZigBee PRO was then developed as an enhancement of the original ZigBee protocol, providing a number of extra features that are particularly useful for very large networks (that may include hundreds or even thousands of nodes).

ZigBee Features

The ZigBee standard builds on the established IEEE 802.15.4 standard for packet-based wireless transport. ZigBee enhances the functionality of IEEE 802.15.4 by providing flexible, extendable network topologies with integrated set-up and routing intelligence to facilitate easy installation and high resilience to failure. ZigBee networks also incorporate listen-before-talk and rigorous security measures that enable them to co-exist with other wireless technologies (such as Bluetooth and Wi-Fi) in the same operating environment.

ZigBee's wireless connectivity means that it can be installed easily and cheaply, and its built-in intelligence and flexibility allow networks to be easily adapted to changing needs by adding, removing or moving network nodes. The protocol is designed such that nodes can appear in and disappear from the network, allowing some devices to be put into a power-saving mode when not active. This means that many devices in a ZigBee network can be battery-powered, making them self-contained and, again, reducing installation costs.

Figure 1 shows a simple example of a ZigBee network in a HVAC (Heating, Ventilation and Air-Conditioning) system.

ZigBee 3.0

ZigBee 3.0 employs the ZigBee PRO protocol and is designed to facilitate general wireless networks that are not market-specific. Thus, devices from different market sectors can belong to the same wireless network - for example, lighting and healthcare devices in a hospital may share a single ZigBee network, allowing data to be routed through any intermediate (routing) device, irrespective of the device functionality. Connecting the network to the Internet brings the devices into the ‘Internet of Things’ (IoT), allowing the network devices to be controlled and monitored from IP-based devices such as computers, tablets and smartphones.

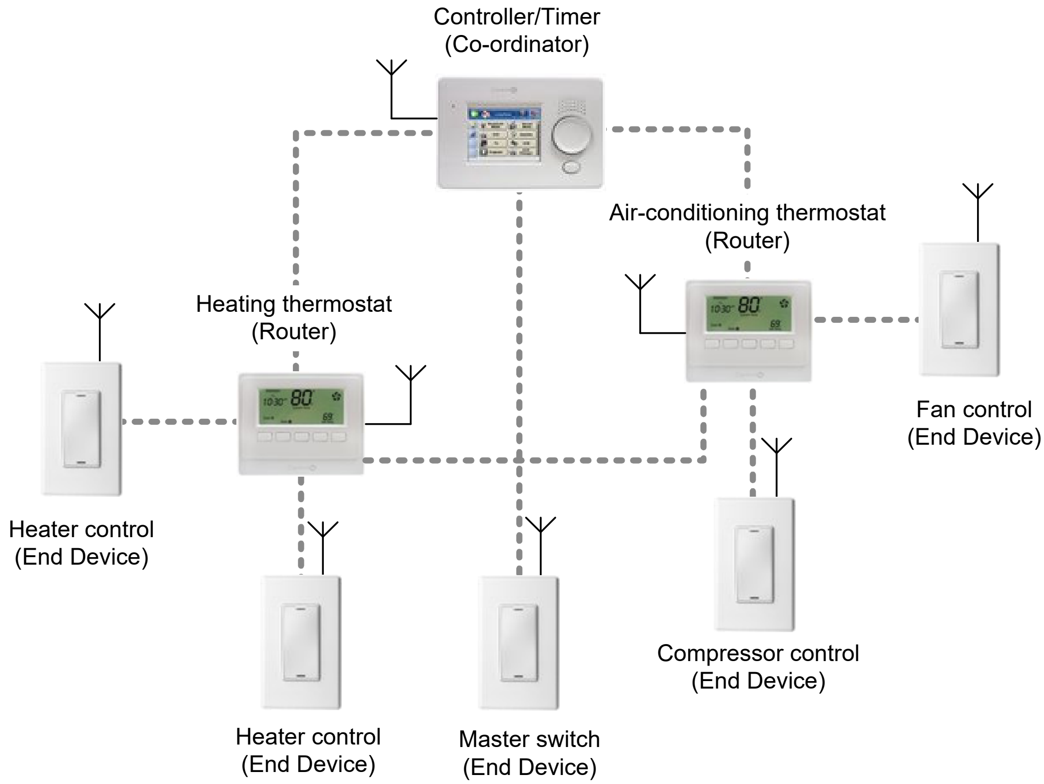


Figure 1: Simple ZigBee Network (Heating and Air-conditioning)

1.1 ZigBee Network Nodes

A wireless network comprises a set of nodes that can communicate with each other by means of radio transmissions, according to a set of routing rules (for passing messages between nodes). A ZigBee wireless network includes three types of node:

- **Co-ordinator:** This is the first node to be started and is responsible for forming the network by allowing other nodes to join the network through it. Once the network is established, the Co-ordinator has a routing role (is able to relay messages from one node to another) and is also able to send/receive data. Every network must have one and only one Co-ordinator.
- **Router:** This is a node with a routing capability, and is also able to send/receive data. It also allows other nodes to join the network through it, so plays a role in extending the network. A network may have many Routers.
- **End Device:** This is a node which is only capable of sending and receiving data (it has no routing capability). A network may have many End Devices.

The deployment of these node types in a ZigBee PRO network is described in Section 1.2. More detailed information about the node types is provided in Section 2.2.1.

1.2 ZigBee PRO Network Topology

ZigBee facilitates a range of network topologies from the simplest Star topology, through the highly structured Tree topology to the flexible Mesh topology. ZigBee PRO is designed primarily for Mesh networks.

A Mesh network has little implicit structure. It is a collection of nodes comprising a Co-ordinator and a number of Routers and/or End Devices, where:

- Each node, except the Co-ordinator, is associated with a Router or the Co-ordinator - this is the node through which it joined the network and is known as its 'parent'. Each parent may have a number of 'children'.
- An End Device can only communicate directly with its own parent.
- Each Router and the Co-ordinator can communicate directly with any other Router/Co-ordinator within radio range.

It is the last property above that gives a Mesh network its flexibility and efficiency in terms of inter-node communication. A Mesh network is illustrated in the figure below.

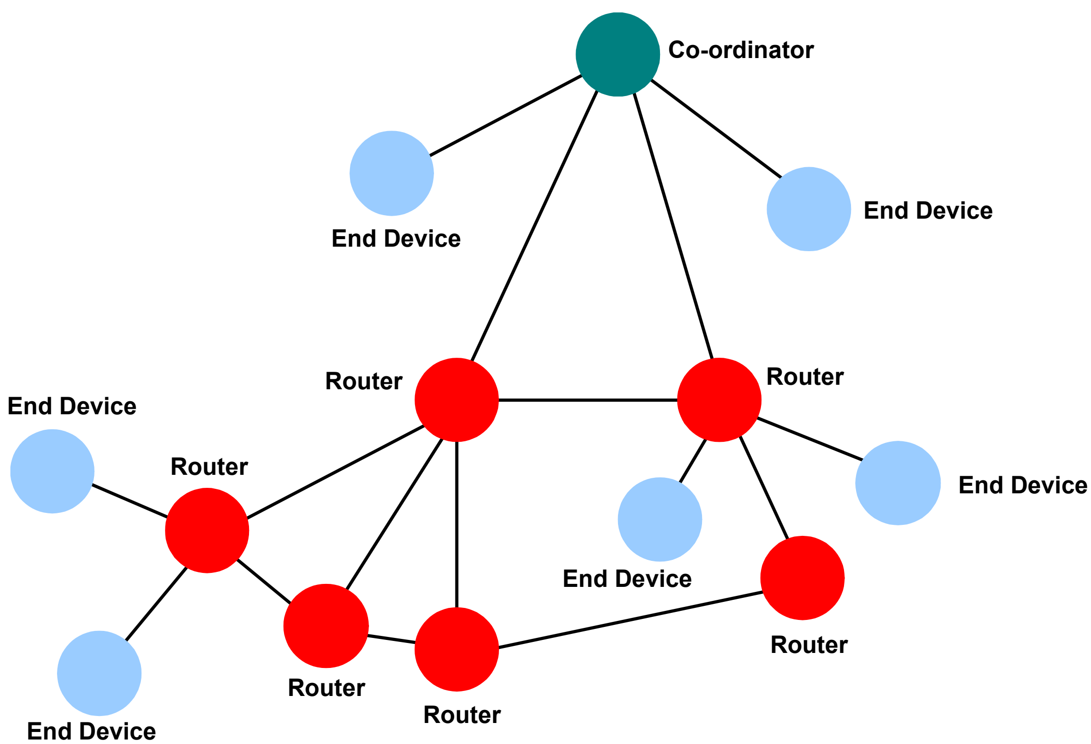


Figure 2: Simple Mesh Network

Mesh networks and their constituent nodes are described in more detail in Section 2.2.2.

1.3 Ideal Applications for ZigBee

ZigBee is suitable for a wide range of applications, covering both commercial and domestic use, which include:

- Point-to-point cable replacement (e.g. wireless mouse, remote controls, toys)
- Security systems (e.g. fire and intruder)
- Environmental control (e.g. heating and air-conditioning)
- Hospital patient monitoring
- Lighting control
- Home automation (e.g. home entertainment, doors, gates, curtains and blinds)
- Automated meter reading (AMR)
- Industrial automation (e.g. plant monitoring and control)

ZigBee's wireless communications also enable some applications to be developed that currently cannot be implemented with cabled systems. Examples are applications that involve mobility, which must be free of cabling (e.g. long-term health monitoring, asset tracking in warehouses). Existing applications (such as lighting control and industrial plant monitoring) that currently rely on cable-based systems can be implemented more cheaply as ZigBee reduces or removes cable installation costs. ZigBee can also be beneficial in environments where cable-based solutions can be difficult and expensive to install - for example, in home security systems, sensors need to be easy to install (no cables or power supply wiring), small and self-contained (battery-powered).

1.4 Wireless Radio Frequency Operation

The IEEE 802.15.4 protocol, on which ZigBee is built, provides radio-based network connectivity operating in one of three possible RF (Radio Frequency) bands: 868, 915 or 2400 MHz. These bands are available for unlicensed use, depending on the geographical area (check your local radio communication regulations).

The characteristics of these RF bands are shown in the table below.

RF Band	Number of Channels
863-876 MHz	63
915-921 MHz	27
Total	90

Table 1: Total number of channels

The channels are distributed across four pages.

Channel Page	Description
863-876 MHz	63
915-921 MHz	27
Total	90

Table 2: Channel distribution across pages

The internal representation of the channels in our stack is as follows:

- A 32-bit mask is used to represent the channel mask.
- The top 5 bits are used for page number and the lower 27 bits are the channel masks.

In 2.4G, page number is 0 channel range 11-26. Thus, it will be 0x00000800 (page 0, Channel 11).

In Sub Gig Page 28 channel 0, is 0xE0000001.

The 868- and 915-MHz bands offer certain advantages such as fewer users, less interference, and less absorption and reflection, but the 2400-MHz band is far more widely adopted for a number of reasons:

- Worldwide availability for unlicensed use
- Higher data rate (250 kbps) and more channels
- Lower power (transmit/receive are on for shorter time due to higher data rate)
- Band more commonly understood and accepted by the marketplace

Therefore, the ZigBee standard assumes operation in the 2400-MHz band, although it is possible to implement ZigBee networks in the other IEEE 802.15.4 bands.

ZigBee includes measures to avoid interference between radio communications. One is its ability to automatically select the best frequency channel at initialisation. It is also possible to adapt to a changing RF environment by moving the network to another channel, if the current channel proves problematic - this ‘frequency agility’ is a core feature of ZigBee PRO. Other measures are described in Section 1.7.

The range of a radio transmission is dependent on the operating environment - for example, indoors or outdoors. Using an NXP JN518x or K32W041/K32W061 standard module fitted with an external dipole antenna, a range of over 1 km can typically be achieved in an open area, but inside a building this can be reduced due to absorption, reflection, diffraction and standing wave effects caused by walls and other solid objects. A high-power module (greater than 15 dBm output power) can achieve a range which is a factor of five greater than that of a standard module. In addition, the range between devices can be extended in a ZigBee network since the network topology (see Section 2.2.2) can use intermediate nodes (Routers) as stepping stones when passing data to destinations.

1.5 Battery-Powered Components

There are many wireless applications that benefit from battery power, including light-switches, active tags and security detectors. The ZigBee and IEEE 802.15.4 protocols are specifically designed for battery-powered applications. From a user perspective, battery power has certain advantages:

- **Easy and low-cost installation of nodes:** No need to connect node to separate power supply
- **Flexible location of nodes:** Nodes can be installed in difficult places where there is no power supply, and can even be used as mobile devices
- **Easily modified network:** Nodes can easily be added or removed, on a temporary or permanent basis

Since these devices are generally small, they use low-capacity batteries and therefore battery use must be optimised. This is achieved by restricting the amount of time for which energy is required by the device.

- Since the major power drain in the system is the operation of the radio, data may be transmitted infrequently (perhaps once per hour or even once per week), which results in a low duty cycle (transmission time as proportion of time interval between transmissions).
- When data is not being sent, the device may revert to a low-power ‘sleep’ mode to minimise power consumption.

In practice, not all nodes in a network can be battery-powered, notably those that need to be switched on all the time for routing purposes (and therefore cannot sleep). These devices can often be installed in a mains-powered appliance that is permanently connected to the mains supply (even if not switched on) - for example, a ceiling lamp or an electric radiator. This avoids the need to install a dedicated mains power connection for the node. Only End Devices are normally battery-powered.

Note: A network device can also potentially use "energy harvesting" to absorb and store energy from its surroundings - for example, the use of a solar cell panel on a device in a well-lit environment.

1.6 Easy Installation and Configuration

One of the great advantages of a ZigBee network is the ease with which it can be installed and configured.

As already mentioned, the installation is simplified and streamlined by the use of certain battery-powered devices with no need for power cabling. In addition, since the whole system is radio-based, there is no need for control wiring to any of the network devices. Therefore, ZigBee avoids much of the wiring and associated construction work required when installing cable-based networks.

The configuration of the network depends on how the installed system has been developed. There are three system possibilities: pre-configured, self-configuring and custom.

- **Pre-configured system:** A system in which all parameters are configured by the manufacturer. The system is used as delivered and cannot readily be modified or extended. Examples: vending machine, patient monitoring unit.
- **Self-configuring system:** A system that is installed and configured by the end-user. The network is initially configured by sending "discovery" messages between devices. Some initial user intervention is required to set up the devices - for example, by pressing buttons on the nodes. Once installed, the system can be easily modified or extended without any re-configuration by the user - the system detects when a node has been added, removed or simply moved, and automatically adjusts the system settings. Example: off-the-shelf home security or home lighting system in which extra devices can be added later.
- **Custom system:** A system that is adapted for a specific application/location. It is designed and installed by a system integrator using custom network devices. The system is usually configured using a software tool.

As indicated above, system commissioning (individually configuring the network nodes) can be performed either using an IO interface (e.g. buttons or a keypad) on the node in a self-configuring system or using a commissioning tool (e.g. run on a lap-top PC) which interacts with the node in a custom system. In the latter case, ZigBee PRO allows commissioning to be conducted in a secure way - for example, using a security key to gain access to the configurable parameters of the node, and using encryption in any wireless communication between the commissioning tool and the node. For more information on system security, refer to Section 1.8.

1.7 Highly Reliable Operation

ZigBee and IEEE 802.15.4 employ a range of techniques to ensure reliable communications between network nodes - that is, to ensure communications reach their destinations uncorrupted. Corruption could result, for example, from radio interference or poor transmission/reception conditions.

- **Data Coding:** At a first level, a coding mechanism is applied to radio transmissions. The coding method employed in the 2400-MHz band uses QPSK (Quadrature Phase-Shift Keying) modulation with conversion of 4-bit data symbols to 32-bit chip sequences. Due to this coding, there is a high probability that a message will get through to its destination intact, even if there are conflicting transmissions (more than one device transmitting in the same frequency channel at the same time).
- **Listen Before Send:** The transmission scheme also avoids transmitting data when there is activity on its chosen channel - this is known as Carrier Sense, Multiple Access with Collision Avoidance (CSMA-CA). Put simply, this means that before beginning a transmission, a node will listen on the channel to check whether it is clear. If activity is detected on the channel, the node delays the transmission for a random amount of time and listens again - if the channel is now clear, the transmission can begin, otherwise the delay-and-listen cycle is repeated.
- **Acknowledgements:** Two systems of acknowledgements are available to ensure that messages reach their destinations:
  - **End-to-End:** When a message arrives at its final destination, the receiving device sends an acknowledgement to the source node to indicate that the message has been received. End-to-end acknowledgements are optional.
  - **Next Hop:** When a message is routed via intermediate nodes to reach its destination, the next routing node (or 'next hop' node) in the route sends an acknowledgement to the previous node to indicate that it has received the message. Next-hop acknowledgements are always implemented.

  In both cases, if the sending device does not receive an acknowledgement within a certain time interval, it resends the original message (it can resend the message several times until the message has been acknowledged).
- **Frequency Agility:** When a ZigBee network is initially set up, the 'best' channel in the relevant radio band is automatically chosen as the operating channel. This is normally the quietest channel detected in an energy scan across the band, but this may not always remain the quietest channel if other networks that operate in the same channel are introduced nearby. For this reason, ZigBee includes an optional frequency agility facility. If the operating channel becomes too noisy, this feature allows the whole network to be moved to a better channel in the radio band.
- **Route Repair:** Networks that employ a Mesh topology (see Section 1.2) have built-in intelligence to ensure that messages reach their destinations. If the default route to the destination node is down, due to a failed intermediate node or link, the network can 'discover' and implement alternative routes for message delivery. ZigBee PRO is designed for Mesh networks and therefore incorporates "route repair" as a core feature.

The above reliability measures allow a ZigBee network to operate even when there are other ZigBee networks nearby operating in the same frequency band. Therefore, adjacent ZigBee networks will not interfere with each other. In addition, ZigBee networks can also operate in the neighbourhood of networks based on other standards, such as Wi-Fi and Bluetooth, without any interference.

1.8 Secure Operating Environment

ZigBee networks can be made secure - measures can be incorporated to prevent intrusion from potentially hostile parties and from neighbouring ZigBee networks. ZigBee also provides privacy for communication between nodes of the same network.

ZigBee PRO security includes the following features:

- Access control lists
- Key-based encryption of communications
- Frame counters

These security measures are outlined below.

Access Control Lists

An access control list allows only pre-defined 'friendly' nodes to join the network.

Key-based Encryption

A very high-security, 128-bit AES-based encryption system (built into the device as a hardware function) is applied to network communications, preventing external agents from interpreting ZigBee network data.

This encryption is key-based. Normally, the same 'network key' is used for all nodes in the network. However, it is possible to use an individual 'link key' between a given pair of network nodes, allowing communications (possibly containing sensitive data) between the two nodes to be private from other nodes in the same network.

Keys can be pre-configured in nodes in the factory, commissioned during system installation or distributed around a working network from a central 'Trust Centre' node. A Trust Centre manages keys and security policies - for example, changing the network key on all network nodes, issuing link keys for node pairs and restricting the hours in which certain events or interactions can occur. Any node can be nominated as the Trust Centre, but it is by default the Co-ordinator.

A distributed security model can alternatively be used, which does not have a Trust Centre - instead, security is managed by the Router nodes in the network.

Frame Counters

The use of frame counters prevents sending the same message twice, and freshness checking rejects any such repeated messages, preventing message replay attacks on the network. An example of a replay attack would be someone recording the open command for a garage door opener, and then replaying it to gain unauthorised entry into the property. Frame counters are described in more detail in Appendix C.

1.9 Co-existence and Interoperability

ZigBee is an open standard devised by the ZigBee Alliance. Any device designed for use in a ZigBee network must comply with the standard. This ensures "co-existence" and, to a certain extent, "interoperability" of ZigBee devices:

- **Co-existence:** The ability of a device to operate in the same space and radio channel as devices in other wireless networks (which possibly use protocols other than ZigBee) without interfering with them
- **Interoperability:** The ability of a device to operate in the same ZigBee network as devices from other manufacturers - that is, to communicate and function with them

The ZigBee Alliance co-ordinates the compliance issues for products based on the ZigBee protocol. It defines two levels of compliance:

- **ZigBee Compliant Platform (ZCP)** applies to modules or platforms intended as building blocks for use in end-products. All NXP products based on the supported chips are designed to be ZigBee Compliant Platforms. See “Chip Compatibility” on page 18.
- **ZigBee Certified Product** applies to end-products that are built on ZigBee Compliant Platforms, and that use public ZigBee Alliance device types and clusters. After successful completion of the ZigBee Alliance Certification programme, the ZigBee Certified Product logo can be applied to the product.

> **Note:** End-products based on manufacturer-specific device types and clusters can also obtain ZigBee Certified Product status, but such products cannot carry the ZigBee Certified Product logo.

Test service providers are authorised by the ZigBee Alliance to undertake testing and certification. For details of authorised test houses, contact the ZigBee Alliance.

In addition, products using an NXP ZCP must also be checked against the radio regulations of the country or countries where they are to be marketed (these checks can often be performed by the same test house).

1.10 Device Types and Clusters

For the purpose of interoperability (described in Section 1.9), the ZigBee Alliance employs the concepts of a device type and a cluster, which define the functionality of a network node. Clusters and device types are introduced below (but more detailed information can be found in Section 2.4).

> **Note:** The ZigBee 'application profile' (which collects together the device types for a market sector) is not so prevalent in ZigBee 3.0. However, application profiles are still supported for backward compatibility.

1.10.1 Clusters

A cluster is a software entity that encompasses a particular piece of functionality for a network node. A cluster is defined by a set of attributes (parameters) that relate to the functionality and a set of commands (that can typically be used to request operations on the cluster attributes). As an example, a thermostat will use the Temperature Measurement cluster that includes attributes such as the current temperature measurement, the maximum temperature that can be measured and the minimum temperature that can be measured (but the only operations that will need to be performed on these attributes will be reads and writes).

The ZigBee Alliance defines a collection of clusters in the ZigBee Cluster Library (ZCL). These clusters cover the functionalities that are most likely to be used. The NXP implementations of these clusters are provided in the ZigBee 3.0 Software Developer's Kit (SDK) and are described in the *ZigBee Cluster Library User Guide (JN-UG-3132)*.

1.10.2 Device Types

The complete functionality of a network node is determined by its device type. This defines a collection of clusters (some mandatory and some optional) that make up the supported features of the device. For example, the Thermostat device uses the Basic and Temperature Measurement clusters, and can also use one or more optional clusters. A device is an instance of a device type.

A network node can support more than one device type. The application for a device type runs on a software entity called an endpoint and each node can have up to 240 endpoints.

All ZigBee 3.0 nodes must implement the ZigBee Base Device (which does not occupy an endpoint), which handles fundamental operations such as commissioning.

The ZigBee device types and ZigBee Base Device are detailed in the *ZigBee Devices User Guide (JN-UG-3131)*.

2. ZigBee PRO Architecture and Operation

This chapter introduces ZigBee PRO from architectural and operational view-points by describing:

- Basic architecture on which ZigBee PRO is based (Section 2.1)
- Concepts for an understanding of ZigBee PRO at the network level (Section 2.2)
- Process of network formation (Section 2.3)
- Concepts for an understanding of ZigBee PRO at the application level (Section 2.4)
- Features and concepts related to message routing (Section 2.5)
- Features and concepts related to exchanging messages (Section 2.6)
- A detailed view of the ZigBee PRO software architecture (Section 2.7)

2.1 Architectural Overview

This section introduces the basic architecture of the software that runs on a ZigBee PRO network node. The software architecture is built on top of IEEE 802.15.4, an established and proven standard for wireless communication.

From a high-level view, the software architecture of any ZigBee network comprises four basic stack layers: Application layer, Network layer, Data Link layer and Physical layer. The Application layer is the highest level and the Physical layer is the lowest level, as illustrated in the figure below.

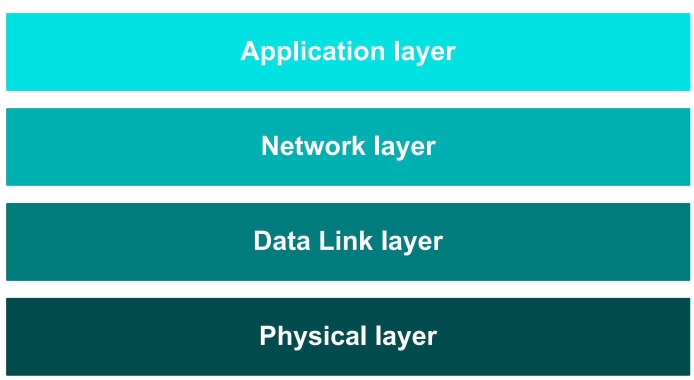


Figure 3: Basic Software Architecture

The basic layers of the ZigBee software stack are described below, from top to bottom:

- **Application layer:** The Application layer contains the applications that run on the network node. These give the device its functionality - essentially an application converts input into digital data, and/or converts digital data into output. A single node may run several applications - for example, an environmental sensor may contain separate applications to measure temperature, humidity and atmospheric pressure.
- **Network layer:** The Network layer provides the ZigBee PRO functionality and the application's interface to the IEEE 802.15.4 layers (see below). The layer is concerned with network structure and multi-hop routing.
- **Data Link layer:** The Data Link layer is provided by the IEEE 802.15.4 standard and is responsible for addressing - for outgoing data it determines where the data is going, and for incoming data it determines where the data has come from. It is also responsible for assembling data packets or frames to be transmitted and disassembling received frames. In the IEEE 802.15.4 standard, the Data Link layer is referred to as IEEE 802.15.4 MAC (Media Access Control) and the frames used are MAC frames.
- **Physical layer:** The Physical layer is provided by the IEEE 802.15.4 standard and is concerned with the interface to the physical transmission medium (radio, in this case), exchanging data bits with this medium, as well as exchanging data bits with the layer above (the Data Link layer). In the IEEE 802.15.4 standard, the Physical layer is referred to as IEEE 802.15.4 PHY.

For a more detailed view of the software architecture of ZigBee PRO, refer to Section Section 2.7.

> **Note:** Security measures are implemented throughout the stack, including the Application layer and lower stack layers.

2.2 Network Level Concepts

This section describes important concepts relating to the work of the ZigBee stack.

2.2.1 ZigBee Nodes

There are three general types of node that can exist in a ZigBee network:

- Co-ordinator
- Router
- End Device

The roles of these node types are described in the sub-sections below.

> **Note:** These roles exist at the network level - a ZigBee node may also be performing tasks at the Application level, independent of the role it plays in the network. For example, a network of ZigBee devices measuring temperature may have a temperature sensor application in each node, irrespective of whether the node is an End Device, Router or the Co-ordinator.

Co-ordinator

All ZigBee networks must have one (and only one) Co-ordinator.

At the network level, the Co-ordinator is mainly needed at system initialisation - it is the first node to be started and performs the following initialisation tasks:

- Selects the frequency channel to be used by the network (usually the one with the least detected activity)
- Starts the network
- Allows child nodes to join the network through it

The Co-ordinator can additionally provide other services such as message routing and security management. It may also provide services at the Application level. If any of these additional services are used, the Co-ordinator must be able to provide them at all times. However, if none of these additional services are used, the network will be able to operate normally even if the Co-ordinator fails or is switched off.

Router

A ZigBee PRO network usually has at least one Router.

The main tasks of a Router are:

- Relays messages from one node to another
- Allows child nodes to join the network through it

Note that a Router cannot sleep, as it must always be available for routing.

End Device

The main tasks of an End Device at the network level are sending and receiving messages. An End Device can only communicate directly with its parent, so all messages to/from an End Device pass via its parent.

An End Device can be battery-powered and, when not transmitting or receiving, can sleep in order to conserve power. Messages destined for a sleep-enabled End Device are buffered by its parent for collection by the End Device once it is awake (also see Section 2.2.2 below).

Note that End Devices cannot relay messages and cannot allow other nodes to connect to the network through them - that is, they cannot have children.

2.2.2 Network Topology

The ZigBee PRO standard was designed to facilitate wireless networks with the Mesh topology.

A Mesh network consists of a Co-ordinator, Routers and End Devices. The Co-ordinator is associated with a set of Routers and End Devices - its children. A Router may then be associated with more Routers and End Devices - its children. This can continue to a number of levels. The relationships between the nodes must obey the following rules:

- The Co-ordinator and Routers can have children, and can therefore be parents.
- A Router can be both a child and a parent.
- End Devices cannot have children, and therefore cannot be parents.

The communication rules for a Mesh network are as follows:

- An End Device can only directly communicate with its parent (and with no other node).
- A Router can directly communicate with its children, with its own parent and with any other Router or Co-ordinator within radio range.
- The Co-ordinator can directly communicate with its children and with any Router within radio range.

The resulting structure is illustrated in Figure 4.

In ZigBee PRO, the maximum depth (number of levels below the Co-ordinator) of a network is 15. The maximum number of hops that a message can make in travelling between the source and destination nodes is 30 (twice the maximum depth).

The ability of a routing node (Router or Co-ordinator) to communicate directly with other routing nodes (within radio range) is the specific property that distinguishes a Mesh network from a Tree network. This property gives rise to very efficient and flexible message propagation, and means that alternative routes can be found if a link fails or there is congestion.

Note that an End Device which is able to sleep is unable to receive messages directly. A message destined for a sleep-enabled End Device is always buffered in its parent node, in case the End Device is asleep when the message arrives. Once the End Device is awake, it must ask or ‘poll’ the parent for messages.

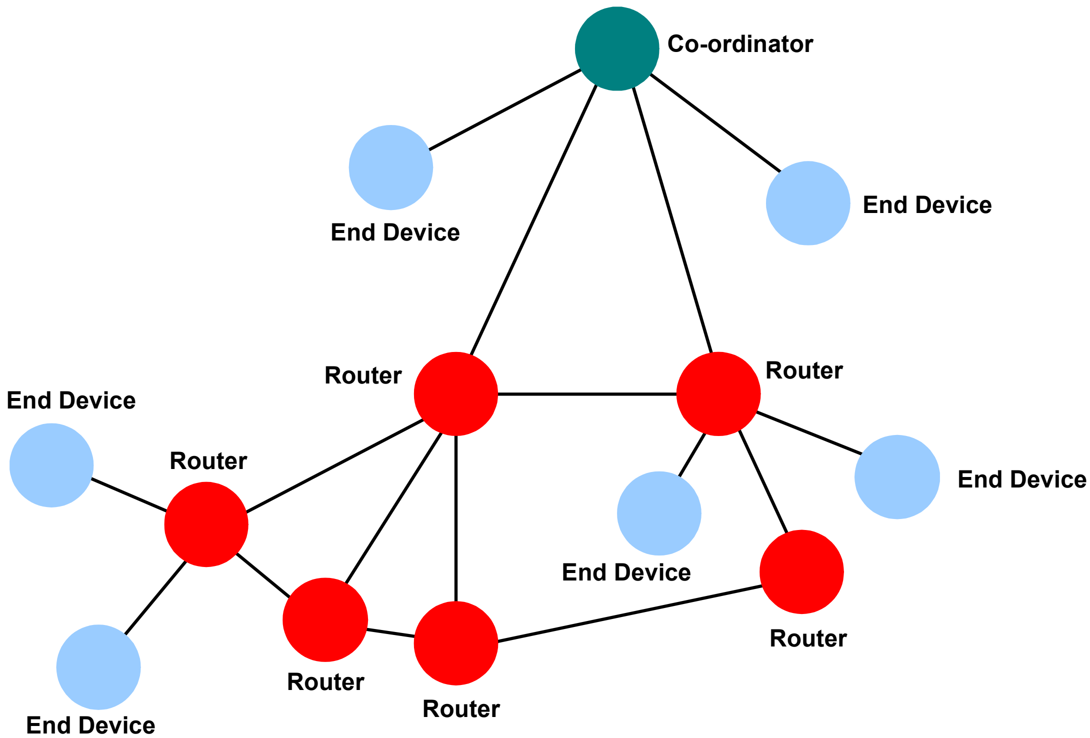


Figure 4: Mesh Topology

In the Mesh topology, a "route discovery" feature is provided which allows the network to find the best available route for a message. Route discovery is described further in Section 2.5.2.

Note that message propagation is handled by the network layer software and is transparent to the application programs running on the nodes.

2.2.3 Neighbour Tables

A routing node (Router or Co-ordinator) holds information about its neighbouring nodes. This information is stored in a Neighbour table containing entries for the node's immediate children, for its own parent and, in a Mesh network, for all peer Routers with which the node has direct radio communication.

It is possible to define the maximum number of entries in a Neighbour table. If this parameter is set to a low value, it will result in a 'long, thin network'.

The structure and configuration of a Neighbour table are described in Appendix B.5.1.

2.2.4 Network Addressing

In a ZigBee network, each node must have a unique identification. This is achieved by means of two addresses:

- **IEEE (MAC) address:** This is a 64-bit address, allocated by the IEEE, which uniquely identifies the device - no two devices in the world can have the same IEEE address. It is often referred to as the MAC address and, in a ZigBee network, is sometimes called the 'extended' address.
- **Network address:** This 16-bit address identifies the node in the network and is local to that network (thus, two nodes in separate networks may have the same network address). It is sometimes called the 'short' address.

  In ZigBee PRO, the network address of a node is dynamically assigned as a random 16-bit value by the parent when the node first joins the network. Due to the randomness of the address allocation, this is known as stochastic addressing. Although random, the parent ensures that the chosen address has not already been assigned to one of its neighbours. In the unlikely event of the address already existing in the network beyond the immediate neighbourhood, a mechanism exists to automatically detect and resolve the conflict. The allocated network address can be retained by the joining node, even if it later loses its parent and acquires a new parent.

  The Co-ordinator always has the network address 0x0000.

While an application on a node may use IEEE/MAC addresses or network addresses to identify remote nodes, the ZigBee PRO stack always uses network addresses for this purpose. To facilitate translation between IEEE/MAC addresses and network addresses, an Address Map table may be maintained on the node, where each table entry contains the pair of addresses for a remote node.

In the NXP implementation of ZigBee PRO, the IEEE/MAC addresses (of other network nodes) are stored in a single place on a node, called the MAC Address table. This avoids the need to repeat the 64-bit IEEE/MAC addresses in other tables, such as the Address Map table and Neighbour table, and therefore saves storage space. Instead, a 16-bit index to the relevant entry in the MAC Address table is stored in the other tables.

It is also possible to define a 16-bit 'group address' which refers to a set of applications (or endpoints - see Section 2.4.1) that may be located across several nodes. Specifying a group address in a data transfer will result in the data being broadcast to all nodes in the network but, at the destinations, the data will only be passed to those applications which are covered by the group address. Refer to Section 5.3 for more details of using group addresses.

2.2.5 Network Identity

A ZigBee network must be uniquely identifiable. This allows more than one ZigBee network to operate in close proximity - nodes operating in the same space must be able to identify which network they belong to.

For this purpose, ZigBee uses two identifiers, as follows:

- **PAN ID:** A 16-bit value called the PAN ID (Personal Area Network Identifier) is used in inter-node communications (implemented at the IEEE 802.15.4 level of the stack) to identify the relevant network. A value for the PAN ID is selected at random by the Co-ordinator when the network is started. When other nodes join the network, they learn the network's PAN ID and use it in all subsequent communications with the network.

  It is possible that the PAN ID generated for a newly installed network will clash with the PAN ID of another network already operating on the same radio channel, in the same neighbourhood. In this case, ZigBee PRO automatically resolves such a conflict by generating another random PAN ID for the new network until a value is obtained that does not clash with the PAN ID of any other detectable network.

- **Extended PAN ID:** A 64-bit value called the Extended PAN ID (EPID) is used in forming the network and subsequently modifying the network, if necessary. This identifier can be pre-set to a random value in the user application that runs on the Co-ordinator. Alternatively, the identifier can be pre-set to zero, in which case the Co-ordinator will adopt its own 64-bit IEEE/MAC address as the Extended PAN ID when the network starts - this is a sure way of obtaining a globally unique value (see Section 2.2.4).

  When a Router or End Device first tries to find a network to join, it will use the Extended PAN ID in either of following ways:

  - If an Extended PAN ID has been pre-set in the user application for the Router or End Device, the node will join the network which has this Extended PAN ID (provided this network is detected).
  - If there is no pre-set Extended PAN ID for the Router or End Device, the node will join the first network detected, irrespective of the Extended PAN ID. The joining node will then learn the Extended PAN ID of its network and later use this identifier to rejoin the network if, for some reason, it loses contact with the network (the node is orphaned).

  For more information on joining a network, refer to Section 2.3.2.

> **Note:** At the Application level, you only need to be concerned with the Extended PAN ID, as the allocation and use of the PAN ID is transparent to the application.

2.3 Network Creation

This section outlines the process of starting and forming a ZigBee PRO network:

- Section 2.3.1 describes how the Co-ordinator starts a network.
- Section 2.3.2 describes how a Router or End Device joins a network as part of the network formation process.

Note: The network formation actions described in this section are performed automatically by the ZigBee stack. The actions required at the application level are described later in Section 5.1.

2.3.1 Starting a Network (Co-ordinator)

The Co-ordinator is responsible for starting a network. It must be the first node to be started and, once powered on, goes through the following network initialisation steps:

1. **Set EPID and Co-ordinator address**

   The Co-ordinator first sets the Extended PAN ID (EPID) for the network and the device's own network address:

   - Sets the EPID to the 64-bit value specified in the Co-ordinator's application (if this value is zero, the EPID will be set to the 64-bit IEEE/MAC address of the Co-ordinator device)
   - Sets the 16-bit network address of the Co-ordinator to 0x0000

2. **Select radio channel**

   The Co-ordinator then selects the radio channel in which the network will operate, within the chosen RF band. The Co-ordinator performs an Energy Detection Scan in which it scans the RF band to find a quiet channel (the scan can be programmed to 'listen' to specific channels). The channel with the least detected activity is chosen.

3. **Set the PAN ID of the network**

   Once the radio channel has been selected, the Co-ordinator chooses a 16-bit PAN ID for the network. To do this, it listens in the channel for traffic from other networks and identifies the PAN IDs of these networks (if any). To avoid conflicts, the Co-ordinator assigns its own network a random PAN ID that is not in use by another network.

4. **Receive join requests from other devices**

   The Co-ordinator is now ready to receive requests from other devices (Routers and End Devices) to wirelessly connect to the network through it. For more information on joining a network, refer to Section 2.3.2.

2.3.2 Joining a Network (Routers and End Devices)

Routers and End Devices can join an existing network already created by a Co-ordinator. The Co-ordinator and Routers have the capability to allow other nodes to join the network through them. The join process is as follows:

1. **Search for network**

   The new node first scans the channels of the relevant RF band to find a network. Multiple networks may operate, even in the same channel, and the selection of a network is the responsibility of the application (for example, this decision could be based on a pre-defined Extended PAN ID).

2. **Select parent**

   The node now selects a parent node within the chosen network by listening to network activity. The node may be able to 'hear' multiple Routers and the Co-ordinator from the network. Given a choice of parents, the node chooses the parent with the smallest depth in the network - that is, the parent closest to the Co-ordinator (which is at depth zero).

3. **Request joining**

   The node sends a message to the desired parent, asking to join the network.

4. **Receive response**

   The node now waits for a response from the potential parent, which determines whether the node is a permitted device and whether the parent is currently allowing devices to join. To determine whether the joining node is a permitted device, the parent consults the Trust Centre (if it is not the Trust Centre itself). If these criteria are satisfied, the parent will then allow the node to join the network as its child. In its acceptance response to its new child, the parent will include the 16-bit network address that it has randomly allocated to the child (see Section 2.2.4).

   If the potential parent is unable to accept the node as a child, a rejection response will be sent to the node, which must then try another potential parent (or another network).

5. **Learn network IDs**

   The new node learns the PAN ID and Extended PAN ID of the network, as well as the network address that it has been assigned. It will need the PAN ID for communications with the network and will need the Extended PAN ID if, at some point in the future, it needs to rejoin the network (it will also be able to re-use its network address if it later rejoins the network).

A Router or Co-ordinator can be configured to have a time-period during which joins are allowed, controlled by its ‘permit joining’ status. The join period may be initiated by a user action, such as pressing a button. An infinite join period can also be set, so that child nodes can join the parent node at any time.

> **Note:** When an orphaned node attempts to rejoin the network, the ‘permit joining’ status of a potential parent is ignored. Thus, the node is able to rejoin the network through a parent on which ‘permit joining’ is disabled.

2.4 Application Level Concepts

This section describes some key concepts required at the application level.

2.4.1 Multiple Applications and Endpoints

A node may have several applications running on it - for example, a node in a smart home network may incorporate an occupancy sensor and a light switch, each of which is an application. In fact, each application implements a ZigBee device type (see Section 1.10). Access to application instances is provided through endpoints, which act as communication ports for the applications.

In order to direct a message to the appropriate application instance on a node, the relevant endpoint must be specified. Endpoints are numbered from 1 to 240. Therefore, to communicate with a remote application instance in a ZigBee network, you need to supply the address of the remote node together with the required endpoint number on the node.

Endpoint 255 is the broadcast endpoint number - the same data can be sent to all application instances on a node by sending the message to this endpoint number.

2.4.2 Descriptors

An application may need to obtain information about the nodes of the network in which it runs, as described in Section 2.4.6. For this, it uses information stored in descriptors in the nodes.

There are three mandatory descriptors and two optional descriptors stored in a node. The mandatory descriptors are the Node, Node Power and Simple descriptors, while the optional descriptors are called the Complex and User descriptors

For each node, there is only one Node and Node Power descriptor, but there is a Simple descriptor for each endpoint. There may also be Complex and User descriptors in the device.

The Node, Node Power and Simple descriptors are outlined below. For full details of the descriptors, refer to Section 8.2.1.

2.4.2.1 Simple Descriptor

The Simple descriptor for an application includes:

- The endpoint on which the application runs and communicates
- The ZigBee device type that the application implements
- The ZigBee clusters that the device type implements
- Whether there are corresponding Complex and User descriptors
- Lists of input and output clusters (see Section 2.4.1) that the application uses and provides, respectively

2.4.2.2 Node Descriptor

The Node descriptor contains information on the capabilities of the node, including:

- Type (End Device, Router or Co-ordinator)
- Frequency band in use (868 MHz, 902 MHz or 2400 MHz)
- IEEE 802.15.4 MAC capabilities - that is, whether:
  - the device can be a PAN Co-ordinator
  - the node implements a Full-Function or Reduced-Function IEEE 802.15.4 device
  - the device is mains powered
  - the device is capable of using MAC security
  - the receiver stays on during idle periods
- Manufacturer code
- Stack compliance revision (of the ZigBee PRO Core specification to which the stack complies - prior to Revision 22/ZigBee2017, these bits were reserved and set to zero)
- Maximum buffer size (the largest data packet that can be sent by an application in one operation)

2.4.2.3 Node Power Descriptor

The Node Power descriptor contains information on how the node is powered:

- Power mode - whether the device receiver is on all the time, or wakes up periodically as determined by the network or only when an application requires (e.g. button press)
- Available power sources - indicates whether the mains supply, or rechargeable or disposable batteries (or any combination) can be used to power the device
- Current power sources - indicates which power source (mains supply, or rechargeable or disposable batteries) is currently being used to power the device
- Current power source level - indicates the level of charge of the current power source

2.4.3 Application Profiles

One of the aims of ZigBee 3.0 is to unify the market-specific ZigBee application profiles that collect together related device types. Application profile identifiers are still needed in ZigBee 3.0 (this ensures backward compatibility with earlier ZigBee versions) but there has been some consolidation of the identifiers - for example, ZigBee Light Link and Home Automation are both covered by the application profile ID 0x0104, which now corresponds to the ZigBee Lighting and Occupancy (ZLO) devices. Profile matching rules exist and are detailed in the ZigBee 3.0 specification.

2.4.4 Device Types

To ensure the interoperability of ZigBee nodes from different manufacturers, the ZigBee Alliance has defined a set of standard device types. A device type (e.g. Dimmable Light) is a software entity which defines the functionality of a device. This functionality is itself defined by the clusters included in the device type, where each cluster corresponds to a specific functional aspect (e.g. Level Control) of the device. For more information on clusters, refer to Section 2.4.5.

A device is an instance of a device type and is implemented by an application that runs on an endpoint. A device type usually supports both mandatory and optional clusters, so a device can be customised in terms of the optional clusters used. The device type implemented by an application is specified in the application's Simple Descriptor (see Section 2.4.2.1). A node may implement more than one device type, each corresponding to a device application that runs on its own endpoint.

Every ZigBee 3.0 node must employ the ZigBee Base Device, which provides a framework for using ZigBee device types and handles fundamental operations such as commissioning (this device does not need an endpoint).

The NXP implementations of the ZigBee device types and ZigBee Base Device are described in the *ZigBee Devices User Guide (JN-UG-3131)*.

2.4.5 Clusters and Attributes

A data entity (e.g. temperature measurement) handled by a ZigBee endpoint is referred to as an attribute. The application may communicate via a set of attributes - for example, a thermostat application may have attributes for temperature, minimum temperature, maximum temperature and tolerance.

ZigBee applications use the concept of a "cluster" for communicating attribute values. A cluster comprises a set of related attributes together with a set of commands to interact with the attributes - for example, commands for reading the attribute values.

A cluster corresponds to a specific piece of functionality for a device application. The total functionality for the application is determined by the ZigBee device type that it implements and the clusters that the device type uses (see Section 2.4.4). Thus, clusters are the functional building blocks of devices.

A cluster has two aspects, which are respectively concerned with receiving and sending commands (one or both aspects may be used by a ZigBee application):

- **Input Cluster or Server Cluster:** This side of a cluster is used to store attributes and receive commands to manipulate the stored attributes (to which the cluster may return responses) - for example, an input cluster would store a temperature measurement and associated attributes, and respond to commands which request readings of these attributes.
- **Output Cluster or Client Cluster:** This side of a cluster is used to manipulate attributes in the corresponding input cluster by sending commands to it (and receiving the responses). Normally, these are write commands to set attribute values and read commands to obtain attribute values (the read values being returned in responses).

The Output/Client and Input/Server sides of a cluster are illustrated below in Figure 5.

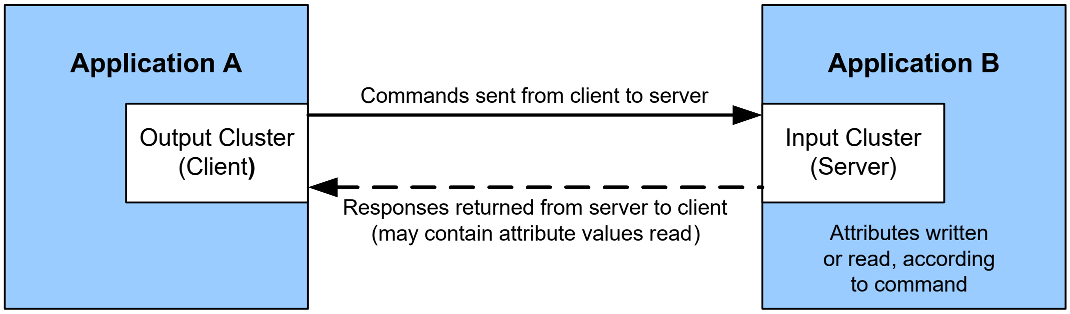


Figure 5: Input (Server) and Output (Client) Clusters

The input clusters and output clusters communicated via an endpoint are listed (separately) in the endpoint's Simple descriptor (see Section 2.4.2.1).

For consistency and interoperability, the ZigBee Alliance have defined a number of standard clusters for different functional areas. These are collected together in the ZigBee Cluster Library (ZCL). Thus, developers can use standard clusters from the ZCL in their device applications. The ZCL is fully detailed in the *ZigBee Cluster Library Specification (075123)* from the ZigBee Alliance. The NXP implementation of these clusters is detailed in the *ZigBee Cluster Library User Guide (JN-UG-3132)*.

A Default cluster (with ID of 0xFFFF) is also available. If the Default cluster is present on an endpoint and a message is received which is destined for a cluster that is not in the endpoint's list of supported input clusters, this message will still be passed to the application (provided it comes from a defined application profile). If it is required, the Default cluster must be explicitly added to the endpoint (see Section 12.4.3).

2.4.6 Discovery

The ZigBee specification provides the facility for devices to find out about the capabilities of other nodes in a network, such as their addresses, which types of applications are running on them, their power source and sleep behaviour. This information is stored in descriptors (see Section 2.4.6) on each node, and is used by the enquiring node to adapt its behaviour to the requirements of the network. Discovery is typically used when a node is being introduced into a user-configured network, such as a domestic security or lighting control system. To integrate the device into the network may require the user to start the integration process by pressing a button or similar. The first task is to find out if there are any appropriate devices with which the new node can communicate.

Device Discovery

Device discovery returns information about the addresses of a network node. The retrieved information can be the IEEE/MAC address of the node with a given network address, or the network address of a node with a given IEEE/MAC address. If the node being interrogated is a Router or Co-ordinator, it may optionally supply the addresses of all the devices that are associated with it, as well as its own address. In this way, it is possible to discover all the devices in a network by requesting this information from the Co-ordinator (network address 0x0000) and then using the list of addresses corresponding to the children of the Co-ordinator to launch other queries about their child nodes.

Service Discovery

Service discovery allows a node to request information from a remote node about the remote node's capabilities. This information is stored in a number of descriptors (see Section 2.4.2) on the remote node, and includes:

- The device type and capabilities of the node
- The power characteristics of the node
- Information about each application running on the node
- Optional information such as serial numbers
- Other user-defined information - for example, easily understandable names such as ‘MtgRoomLight’

Requests for these descriptors are made by a device during the discovery process that is typically part of the device's configuration and integration into a ZigBee network.

2.4.7 ZigBee Device Objects (ZDO)

A special application, common to all ZigBee devices, is provided to manage the various processes which have been described. This application is the ZigBee Device Objects or ZDO. It resides in the Application layer of a node, and can communicate with remote nodes via endpoint 0 using the ZigBee Device Profile (ZDP) and associated clusters. It has the following roles:

- Defines the type of network device: Co-ordinator, Router or End Device
- Initialises the node to allow applications to be run
- Performs the device discovery and service discovery processes
- Implements the processes needed to allow a Co-ordinator to create a network, and Routers and End Devices to join and leave a network
- Initiates and responds to binding requests (see Section 2.6.2)
- Provides security services which allow secure relationships to be established between applications
- Allows remote nodes to retrieve information from the node, such as Routing and Binding tables, and to perform remote management of the node, such as instructing it to leave the network

The ZDO uses services within the stack to implement these roles and provides a means of allowing user applications to access stack services.

2.5 Network Routing

The basic operation in a network is to transfer data from one node to another. The data is sourced from an input (possibly a switch or a sensor) on the originating node, and is communicated to another node which can interpret and use the data.

In the simplest data communication, the data is transmitted directly from the source node to the destination node. However, if the two nodes are far apart or in a difficult environment, direct communication may not be possible. In this case, it is necessary to send the data to another node within radio range, which then passes it on to another node, and so on until the desired destination node is reached - that is, to use one or more intermediate nodes as stepping stones. The process of receiving data destined for another node and passing it on is known as routing.

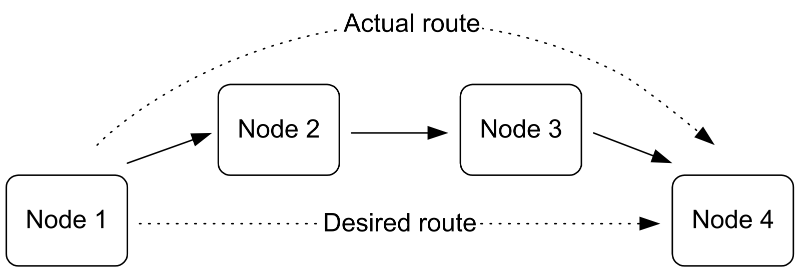


Figure 6: Message Routing

Routing allows the range of a network to be extended beyond the distances supported by direct radio communication. Remote devices can join the network by connecting to a Router.

Note: Application programs in intermediate nodes are not aware of the relayed message or its contents - the relaying mechanism is handled by the ZigBee stack.

2.5.1 Message Addressing and Propagation

If a message sent from one node to another needs to pass through one or more intermediate nodes to reach its final destination (up to 30 such hops are allowed), the message carries two destination addresses:

- Address of the final destination
- Address of the node which is the next "hop"

ZigBee PRO is designed for Mesh networks (see Section 2.2.2) in which the message propagation path (the route) depends on whether the target node is in radio range:

- If the target node is in range, only the "final destination" address is used.
- If the target node is not in range, the "next hop" address is that of the first node in the route to the final destination.

The “next hop” address is determined using information stored in a Routing table on the routing node (Router or Co-ordinator). An entry of this table contains information for a remote node, including the network addresses of the remote node and of the next routing node in the route to the remote node. Thus, when a message is received by a routing node, it looks for the destination address in its Routing table and extracts “next hop” address from this table to insert into the message. The message is then passed on and propagation continues in this way until the target node is reached.

Note that if the message originates from an End Device, the message will always be first passed to the source node’s parent before being passed on.

2.5.2 Route Discovery

The ZigBee stack network layer supports a ‘route discovery’ facility which finds the best available route to the destination, when sending a message. A message is normally routed along an already discovered mesh route, if one exists, otherwise the routing node (Router or the Co-ordinator) involved in sending the message initiates a route discovery. Once complete, the message will be sent along the calculated route.

The mechanism for route discovery between two End Devices has the following steps:

1. A route discovery broadcast is sent by the parent of the source End Device, and contains the destination End Device’s network address.
2. All routing nodes will eventually receive the broadcast, one of which is the parent of the destination End Device
3. The parent of the destination node sends back a reply addressed to the parent of the source node.
4. As the reply travels back through the network, the hop count and a signal quality measure for each hop are recorded. Each routing node in the path can build a Routing table entry containing the best path to the destination End Device.

   The choice of best path is usually the one with the least number of hops, although if a hop on the most direct route has a poor signal quality (and hence a greater chance that retries will be needed), a route with more hops may be chosen.
5. Eventually each routing node in the path will have a Routing table entry and the route from source to destination End Device is established. Note that the corresponding route from destination to source is not known - the route discovered is unidirectional.

A source Router implements route discovery in a similar way to the above except the Router broadcasts its own route discovery message (without needing its parent to do this). Similarly, the Co-ordinator broadcasts its own route discovery messages.

Note: Message routing is performed automatically by the ZigBee stack and is transparent to the user application. If required, route discovery is also automatic and transparent to the application.

2.5.3 'Many-to-one' Routing

A common scenario in a wireless network is the need for most network nodes to communicate with a single node which performs some centralised function, e.g. a gateway. This node is often referred to as a concentrator.

In order to establish communication with the concentrator, each remote node may initiate a 'route discovery', resulting in a corresponding entry in the Routing table of each routing node along the way. If most network nodes need to communicate with the concentrator, many such route discoveries may be initiated. Where the resulting routes have a common leg, the relevant Routing table entries will not be duplicated but shared. However, a large number of simultaneous route discoveries may require significant memory space in the nodes near the concentrator for the temporary storage of route discovery information, and possibly result in memory overflow and traffic congestion.

A more efficient method of establishing routes to a concentrator is for the concentrator to initiate a 'many-to-one' route discovery for routes from all other network nodes to itself. To do this, the concentrator broadcasts a route discovery request and the Routing tables are updated as the broadcast propagates through the network. Since no responses are generated, the temporary storage of route discovery information is not required and network traffic congestion is minimised.

Many-to-one route discovery is illustrated in the figure below.

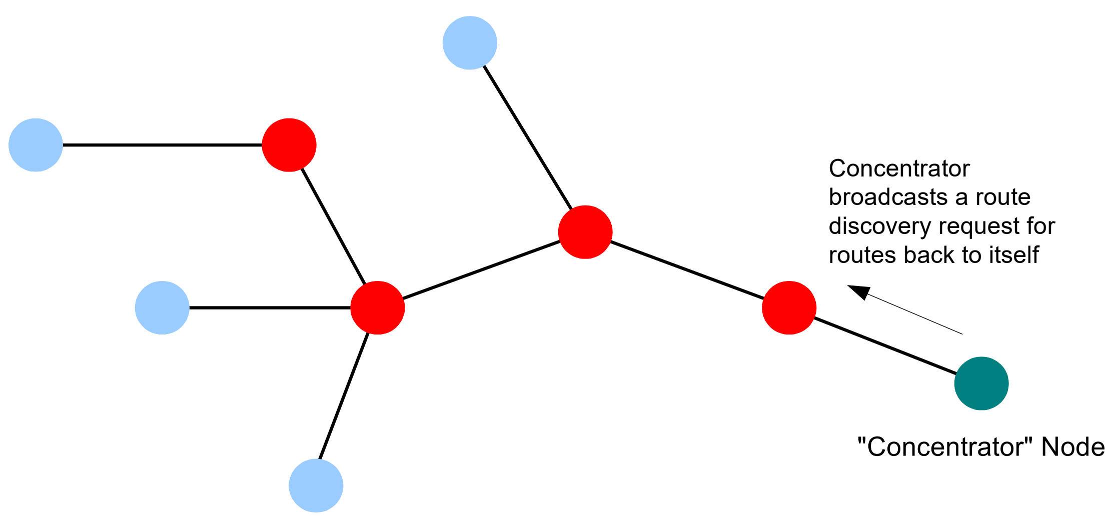


Figure 7: 'Many-to-one' Route Discovery

In order to avoid the storage of return routes (from the concentrator) in the Routing tables of intermediate nodes, the technique of source routing is used - the outward route taken by a message to the concentrator is remembered by the concentrator and embedded in the response message. In this case, the response message must carry up to 30 addresses of the nodes along the return route (maximum number of hops allowed is 30).

2.6 Network Communications

This section considers the processes that are needed to allow a network of devices to exchange information and perform useful functions. In order to communicate with each other, two nodes must be compatible in that one node can produce data which the other node can accept and interpret in a meaningful way. For example, a temperature sensor node produces a temperature measurement that a heating controller node can use to control a central heating system.

When a new node joins a network, it must find compatible nodes with which it is able to communicate - this process is facilitated by the Service Discovery mechanism. It must then choose which of the compatible nodes it will communicate with. A method of pairing nodes for easy communication is provided by the binding mechanism.

Note: While you should always use Service Discovery to find compatible nodes, binding is an optional method for pairing compatible nodes.

Service Discovery and binding are covered in the sub-sections below.

2.6.1 Service Discovery

A device joining a network must be able to find other devices in the network that can use the information it provides, or that can generate the information needed by the device to perform its own function. A node can use Service Discovery to find nodes with which it can communicate. Service Discovery is introduced in Section 2.4.6.

The node requests the required services from other nodes by means of a broadcast message that propagates throughout the network. Any node that has the requested services then uni-casts a response back to the requesting node. This means that the requesting node may receive more than one response.

A response includes the network address of the remote node that contains the requested services. The node stores this address locally and the application can then use the address for all future communications to the remote node. This is referred to as direct addressing.

Alternatively, rather than using direct addressing in their communications, two nodes can communicate through the binding mechanism, described in Section 2.6.2 below.

2.6.2 Binding

Once two nodes have been found to be compatible through Service Discovery (see Section 2.6.1), they may be paired for communication purposes. For example, a light-switch may be paired with a particular light, and we must ensure that this light-switch only ever switches the light that it is intended to control. An easy way to pair nodes for communication is provided by the binding mechanism.

Binding allows nodes to be paired in such a way that a certain type of output data from one node is automatically routed to the paired node, without the need to specify the destination address and endpoint every time. The two nodes must first be bound together using the address and relevant endpoint number for each node - these can be obtained through Service Discovery, described in Section 2.6.1. A binding has a source node and a destination node, relating to the direction in which data will be sent between the nodes (from source to destination). The details of a binding are stored as an entry in a binding table, normally held on the source node of the binding or sometimes on another nominated node.

In order to establish a binding, it must be requested in either of the following ways:

- Binding request is submitted to the source node for the binding by either the source node itself or a remote node (not one of the nodes to be bound).
- Binding requests are submitted to the Co-ordinator by the source and destination nodes for the binding (for example, by pressing a button on each node to generate a binding request). The two binding requests must be received within a certain timeout period.

During the binding process, the Binding table for the source node is updated or, if necessary, created.

Binding occurs at the application level using clusters (described in Section 2.4.5). In order for two applications to be bound, they must support the same cluster.

The binding between two applications is specified by:

- The node address and endpoint number of the source of the binding (e.g. a light-switch)
- The node address and endpoint number of the destination of the binding (e.g. the load controller for a light)
- The cluster ID for the binding

The following types of binding can be achieved:

- **One-to-one:** This is a simple binding in which an endpoint is bound to one (and only one) other endpoint, requiring a single Binding table entry.
- **One-to-many:** This is a binding in which a source endpoint is bound to more than one destination endpoint. The binding is achieved by having multiple Binding table entries for the same source endpoint.
- **Many-to-one:** This is a binding in which more than one source endpoint is bound to a single destination endpoint. The binding is achieved by multiple nodes having one-to-one bindings for the same destination endpoint.

These are illustrated in the figure below.

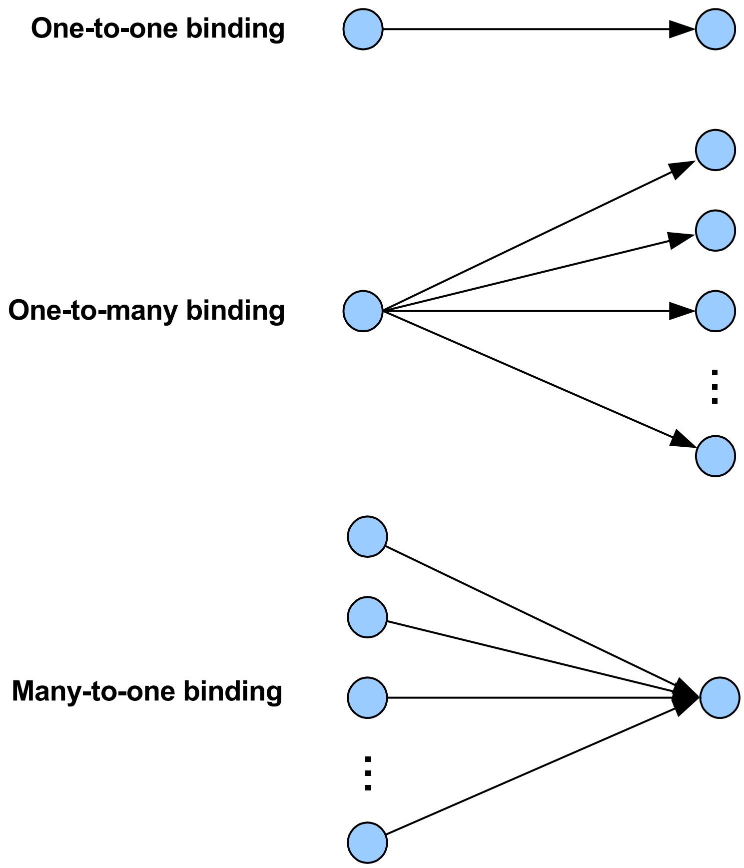


Figure 8: Types of Binding

As an example of these bindings, consider a switch and load controller for lighting:

- In the one-to-one case, a single switch controls a single light
- In the one-to-many case, a single switch controls several lights
- In the many-to-one case, several switches control a single light, such as a light on a staircase, where there are switches at the top and bottom of the stairs, either of which can be used to switch on the light

It is also possible to envisage many-to-many bindings where in the last scenario there are several lights on the staircase, all of which are controlled by either switch.

The way bindings are configured depends on the type of network (described in Section 1.6), as follows:

- **Pre-configured system:** Bindings are factory-configured and stored in the application image.
- **Self-configuring system:** Bindings are automatically created during network installation using discovery software that finds compatible nodes/clusters.
- **Custom system:** Bindings are created manually by the system integrator or installation technician, who may use a graphical software tool to draw binding lines between clusters on nodes.

2.7 Detailed Architecture

This section elaborates on the simplified software architecture presented in Section 2.1 The detailed architecture is illustrated in the figure below.

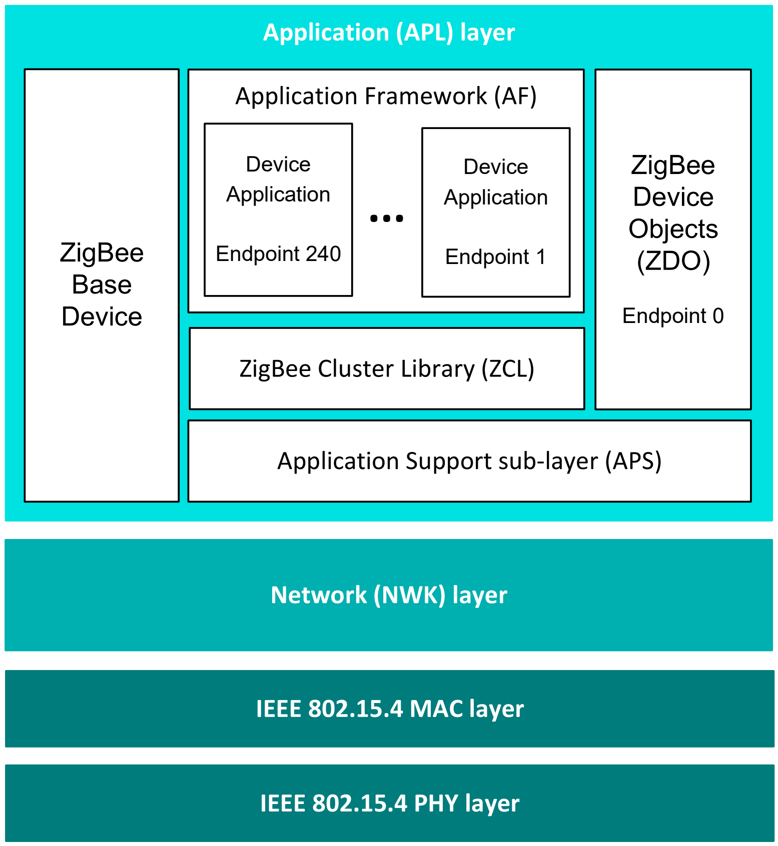


Figure 9: Detailed Software Architecture

2.7.1 Software Levels

The software architecture diagram in Figure 9 shows (from top to bottom):

Application (APL) Layer

This includes:

- **Applications:** Up to 240 application instances may be supported on a single ZigBee node. Each application instance communicates via an endpoint, where endpoints are numbered between 1 and 240 (note that endpoint 0 is reserved for the ZDO of the node - see below).
- **Application Framework (AF):** The AF facilitates interaction between the applications and the APS layer (see below) through an interface known as a Service Access Point or SAP.
- **ZigBee Device Objects (ZDO):** The ZDO represents the ZigBee node type of the device (Co-ordinator, Router or End Device) and has a number of communication roles. The ZDO communicates via endpoint 0. For more information, refer to Section 2.4.7.
- **ZigBee Base Device:** This device is required for all ZigBee 3.0 nodes and deals with essential tasks for the whole node, such as commissioning. It does not occupy an endpoint.
- **ZigBee Cluster Library (ZCL):** The ZCL provides the standard ZigBee clusters used by the device applications that run on the endpoints.
- **Application Support sub-layer (APS):** The APS layer is responsible for:
  - Communicating with the relevant application - for example, when a message arrives to illuminate an LED, the APS layer relays this instruction to the responsible application using the endpoint information in the message.
  - Maintaining binding tables (see Section 2.6.2) and sending messages between bound nodes
  - Providing communication with the Trust Centre to obtain authorisation

  The APS layer has an associated database, called the APS Information Base (AIB). This contains attributes that mainly relate to system security.

Network (NWK) Layer

The NWK layer handles network addressing and routing by invoking actions in the MAC layer. It provides services for:

- Starting the network
- Assigning network addresses
- Adding devices to and removing them from the network
- Routing messages to their intended destinations
- Applying security to outgoing messages
- Implementing route discovery and storing Routing table information

The NWK layer has an associated database, called the NWK Information Base (NIB). This contains attributes required in the management of the NWK layer.

Physical/Data Link Layers

This consists of the IEEE 802.15.4 PHY and MAC layers, described in Section 2.1.

> **Note:** The Security Service Provider (not shown in the figure) spans the APS and NWK layers, providing security services - for example, security key management, datastream encryption and decryption. It may use hardware functions provided in the node to perform the encode and decode operations efficiently.

3. ZigBee Stack Software

This chapter introduces the NXP ZigBee 3.0 stack software.

3.1 Software Overview

The NXP ZigBee 3.0 software provides all components of the ZigBee stack detailed in Section 2.7. In addition, it includes the JN51xx Core Utilities (JCU). The basic architecture of this software, in relation to the wireless network application, is illustrated in the figure below.

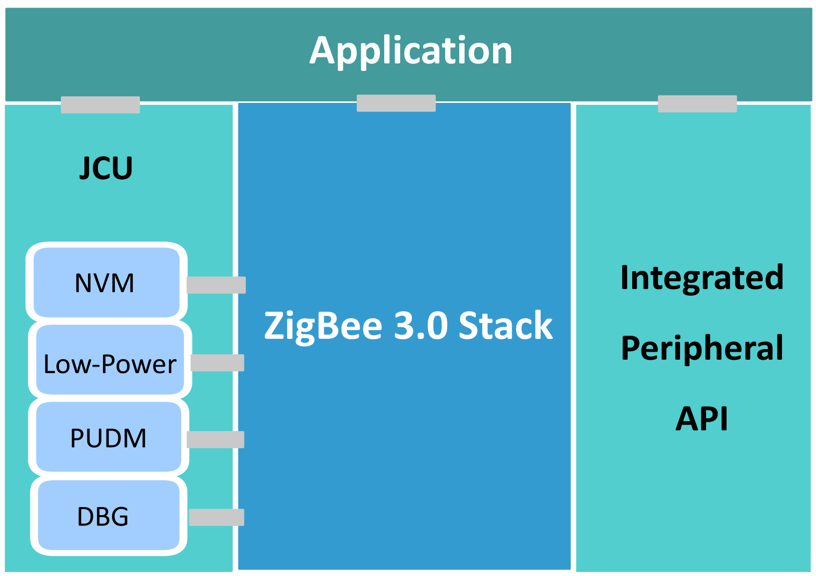


Figure 10: Overview of NXP ZigBee Software Architecture

The NXP ZigBee 3.0 software includes Application Programming Interfaces (APIs) to facilitate simplified application development for wireless networks. These APIs comprise C functions that can be incorporated directly in application code.

Two general categories of API are supplied:

- ZigBee PRO APIs - see Section 3.1.1
- JCU APIs - see Section 3.1.2

The above figure also shows the Integrated Peripherals API that can be used to interact with the on-chip hardware peripherals of the device. This API is described in the MCUXpresso SDK API Reference Manual ((MCUXSDKJN5189APIRM or MCUXSDKK32W041APIRM).

In addition, the ZigBee Cluster Library (ZCL) provides APIs for the individual clusters, as well as more general ZCL functions. The ZCL is located within the stack block.

All the above APIs are supplied in the ZigBee 3.0 Software Developer's Kit (SDK). For more details on the SDK, refer to Section 4.1.

3.1.1 ZigBee PRO APIs

The ZigBee PRO APIs are concerned with network-specific operations and easy interaction with the ZigBee PRO stack from the application code. These C-function APIs are supplied in the ZigBee 3.0 SDK (see Section 4.1).

There are three ZigBee PRO APIs:

- **ZigBee Device Objects (ZDO) API:** Concerned with the management of the local device (e.g. introducing the device into a network)
- **ZigBee Device Profile (ZDP) API:** Concerned with the management of remote devices (e.g. device discovery, service discovery, binding)
- **Application Framework (AF) API:** Concerned with creating data frames for transmission and modifying device descriptors

The locations of these APIs, as well as the JCU and ZCL APIs, are illustrated in the figure below.

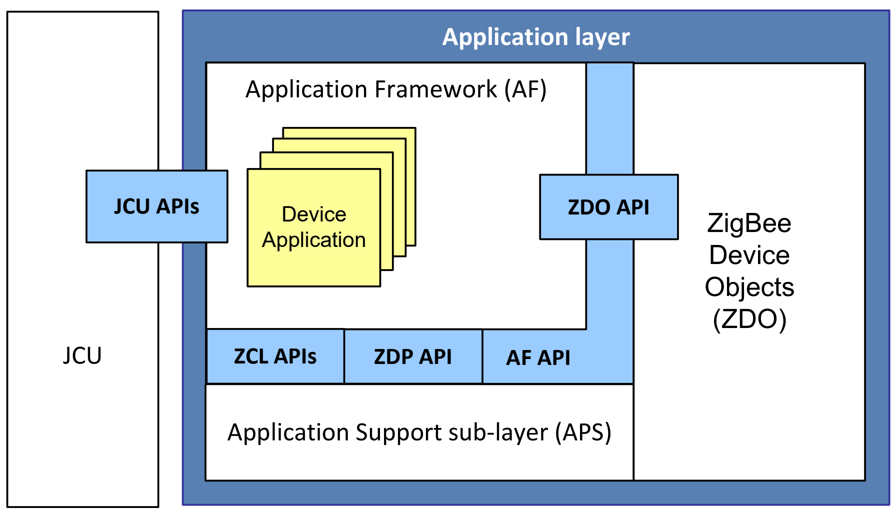


Figure 11: Locations of ZigBee 3.0 APIs

Note: The C functions of all the ZigBee PRO APIs are fully detailed in Part II: Reference Information of this manual.

3.1.2 JCU APIs

The Core Utilities (JCU) provide an easy-to-use interface to simplify the programming of a range of non-network-specific operations. These utilities/modules each have a C function API, which allows a module to be used from a user application. The JCU is supplied in the ZigBee 3.0 SDK.

The JCU modules are outlined below:

- **Non-Volatile Memory Manager (NVM):** This module handles the storage of context and application data in Non-Volatile Memory (NVM), and the retrieval of this data. It provides a mechanism by which the device can resume operation without loss of continuity following a power loss.
- **Low-Power:** This module manages the transitions of the device into and out of low-power modes, such as sleep mode.
- **Protocol Data Unit Manager (PDUM):** This module is concerned with managing memory, as well as inserting data into messages to be transmitted and extracting data from messages that have been received.

> **Note:** The JCU modules are fully described in the *DK6 Core Utilities User Guide (JN-UG-3133)*.

3.2 Summary of API Functionality

This section summarises the roles of the NXP ZigBee PRO and JCU APIs in an application. The table below indicates the APIs needed for the different functionality that your may require in your code:

Functionality	ZigBee PRO APIs	JCU APIs
Essential functionality, including network formation and management	**ZDO API:** Network formation and local network management **ZDP API:** Network discovery and remote network management	
Basic data transfer	**AF API:** Sending and receiving data messages	**PDUM API:** Assembling and disassembling data messages
Binding endpoints for data transfers between them	**ZDO API:** Basic binding **ZDP API:** Manipulation of remote Binding tables	
Low-power modes (Sleep and Doze)		**Low-Power API:** Managing low-power modes
Preserving context data (e.g. for resuming operation after sleep without memory held)		**NVM API:** Saving and restoring context data
Network security	**ZDO API:** Managing security	

Table 3: Use of ZigBee PRO and JCU APIs

Note that:

- ZigBee PRO API function names are prefixed with 'ZPS' (for 'ZigBee PRO Stack' function). The function names also incorporate 'Apl' (for 'Application' function) and the acronym for the API to which the function belongs:
  - ZDO function names include 'Zdo' (e.g. **ZPS_eAplZdoPoll()**)
  - ZDP function names include 'Zdp' (e.g. **eAplZdpActiveEpRequest()**)
  - AF function names include 'Af' (e.g. **ZPS_eAplAfUnicastDataReq()**)
- JCU API function names are prefixed with the acronym for the JCU module to which the function belongs:
  - 'NVM' for NVM functions
  - 'PWR' for Low-Power functions
  - 'PDUM' for PDUM functions

A similar naming convention is used in structures and enumerations.

4. Application Development Overview

This chapter provides an overview of the main phases in developing a ZigBee 3.0 wireless network product. It is important that you refer to this chapter, particularly Section 4.3, before and during your product development.

You will need to develop an application program for each node type in your product - Co-ordinator, Router and End Device. If a node type has variants, you may need to develop a separate application for each variant - for example, an End Device which is a Light Sensor and an End Device which is an On/Off Light Switch in a lighting system.

4.1 Development Environment and Resources

This User Guide supports the NXP ZigBee 3.0 Software Developer's Kits (SDKs) for the JN518x and K32W041/K32W061 devices.

4.1.1 Development Platform

MCUXpresso

NXP MCUXpresso provides an Eclipse-based platform for developing applications for the JN518x and K32W041/K32W061 devices. It can be obtained from https://community.nxp.com/community/mcuxpresso/mcuxpresso-ide and must be a registered edition. **The required version of MCUXpresso is indicated in the Release Notes for the ZigBee 3.0 SDK.**

For installation and operational instructions, first refer to the *MCUXpresso Installation and User Guide*. More detailed operational instructions are provided in the *MCUXpresso User Guide*, available from the above web site.

4.1.2 ZigBee 3.0 SDK

The ZigBee 3.0 SDK provides the stack and API software resources needed to develop ZigBee 3.0 applications for the JN518x and K32W041/K32W061 devices and includes:

- ZigBee PRO and IEEE 802.15.4 stack software
- ZigBee PRO APIs
- ZigBee Base Device Behaviour (BDB) APIs
- ZigBee Cluster Library (ZCL) APIs
- Connectivity Framework APIs
- ZPS Configuration Editor
- Integrated Peripherals APIs and Board APIs

NXP-specific tools have been devised for MCUXpresso, including the ZPS Configuration Editor, which is provided as an Eclipse plug-in. This tool is used to set network parameters and is described in Chapter 12.

This ZigBee 3.0 SDK contains device-specific plug-ins for the MCUXpresso platform.

MCUXpresso must be installed before the ZigBee 3.0 SDK. Refer to Section 4.1.1 for information on this toolchain.

4.2 Support Resources

While developing your ZigBee 3.0 application for a JN518x or K32W041/K32W061 device, you should consult this User Guide along with the:

- *ZigBee Devices User Guide (JN-UG-3131)*
- *ZigBee Cluster Library User Guide (JN-UG-3132)*
- *Connectivity Framework Reference Manual*

Further support in the development of ZigBee 3.0 applications for the JN518x or K32W041/K32W061 devices is provided in the following NXP Application Notes:

- *ZigBee 3.0 Base Device Template (JN-AN-1217)*
- *ZigBee 3.0 Light Bulbs (JN-AN-1244)*
- *ZigBee 3.0 Controller and Switch (JN-AN-1245)*
- *ZigBee 3.0 Sensors (JN-AN-1246)*
- *ZigBee 3.0 IoT Control Bridge (JN-AN-1247)*

Note: The relevant software and documentation resources can be obtained via the Wireless Connectivity area of the NXP web site (for the web address, see "Support Resources" on page 18).

4.3 Development Phases

The main phases of development of a ZigBee 3.0 application are as follows and are conducted in MCUXpresso:

1. **Network Configuration:** Configure the network parameters for the nodes using the ZPS Configuration Editor (refer to Chapter 11 and Chapter 12).
2. **Application Code Development:** Develop the application code for your nodes using the ZigBee PRO, ZCL, BDB and JCU APIs.
3. **Application Build:** Build the application binaries for your nodes.
4. **Node Programming:** Load the application binaries into Flash memory on your nodes.

Note: As a starting point for your application development, you may wish to use one or more of the Application Notes listed in Section 4.2.

5. Application Coding with ZigBee PRO APIs

This chapter outlines how to use functions of the NXP ZigBee PRO APIs to perform common operations required in a ZigBee PRO wireless network application.

The operations covered in this chapter are divided into the following areas:

- Forming a ZigBee PRO wireless network (Section 5.1)
- Discovering the properties of the formed network (Section 5.2)
- Managing group addresses (Section 5.3)
- Binding nodes for easy communication between them (Section 5.4)
- Transferring data between nodes (Section 5.5)
- Leaving and rejoining the network (Section 5.6)
- Return codes and extended error handling (Section 5.7)
- Implementing ZigBee security (Section 5.8)
- Using support software features - message queues and timers (Section 5.9)
- Using advanced features (Section 5.10)

The main stages of the life-cycle of a wireless network are illustrated in Figure 12. These stages incorporate many of the high-level operations described in this chapter.

Many of the functions referenced in this chapter are non-blocking functions that submit a request to the relevant node(s) of the network and then return - these functions have **Request** or **Req** in their names. The recipient of the request will normally reply by sending a response to the node that initiated the request. Once received, this response message can be collected using the function **ZQ_bZQueueReceive()** - see Section 5.9.1.1.

The ZigBee PRO API functions mentioned in this chapter are fully detailed in Part II: Reference Information of this manual.

Tip: Further assistance in developing your own ZigBee 3.0 applications is provided in a range of NXP Application Notes (see Section 4.2).

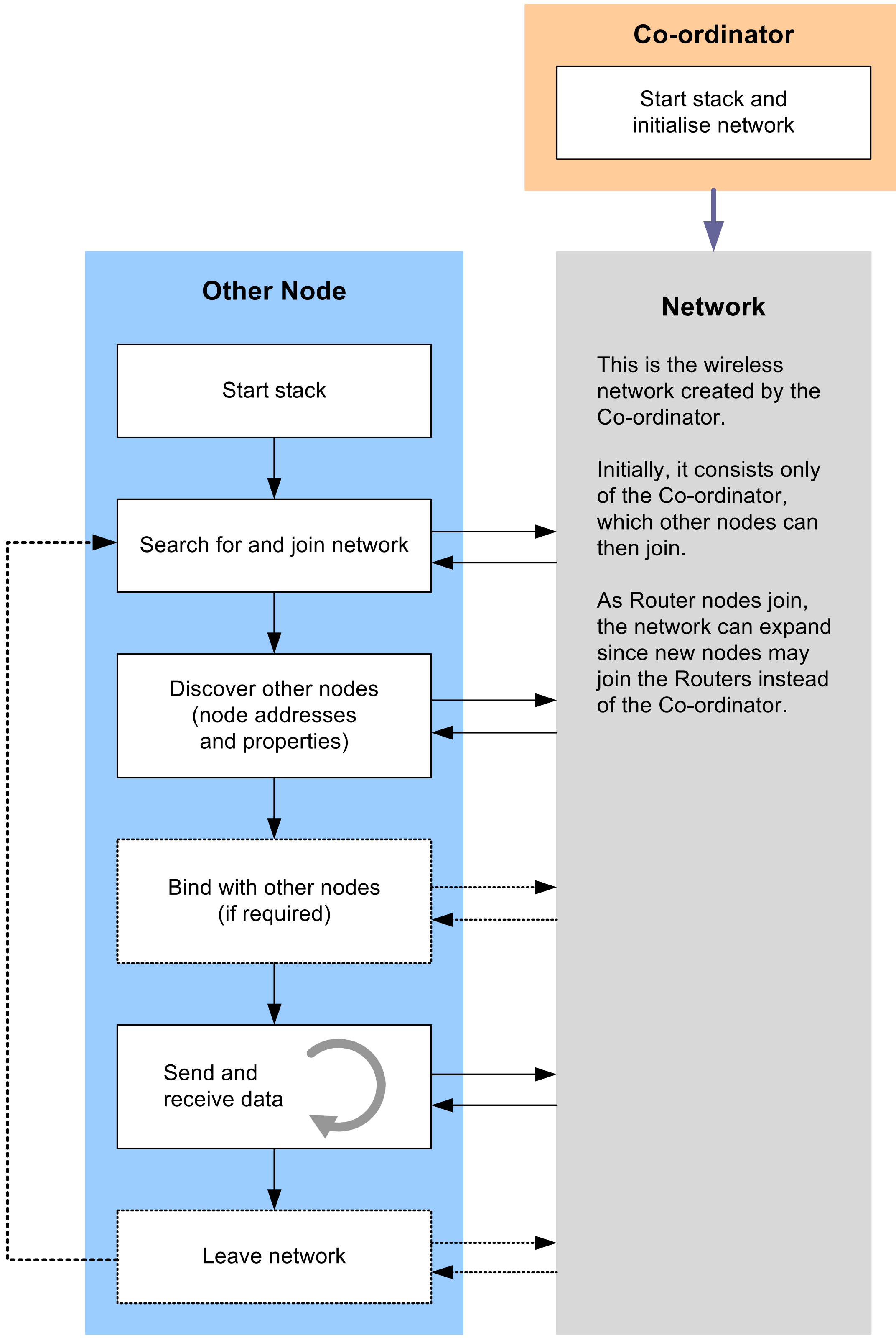


Figure 12: Wireless Network Life-Cycle

5.1 Forming and Joining a Network

This section describes how to form a wireless network by first starting the Co-ordinator and then starting the other nodes, which join the network initiated by the Co-ordinator.

Important: In order to start any network node, certain configuration values must have been pre-set for the application. This configuration is performed using the ZPS Configuration Editor, described in Chapter 12.

At initialisation, the same function calls are needed for all node types (although, once started, the stack will perform initialisation tasks according to the specific node type, as described in Section 5.1.1 and Section 5.1.2). These function calls are listed below, in the required order:

1. **PDUM_vInit()** to initialise the JCU Protocol Data Unit Manager (PDUM)
2. **PWR_Init()** to initialise the Low-Power module in order to facilitate low-power modes such as sleep and doze
3. **NvModuleInit()** to initialise the JCU Non-Volatile Memory Manager (NVM) in order to save context and application data for retrieval after a power break
4. **eZCL_Initialise()** to initialise the ZigBee Cluster Library (ZCL)
5. **eZCL_Register()** for a custom device type, or the equivalent registration function for a standard ZigBee device type, to register an endpoint for the application
6. **zps_eAplAfInit()** to initialise the Application Framework
7. **BDB_vInit()** to initialise the ZigBee Base Device
8. **zps_eAplZdoStartStack()** to start the ZigBee PRO stack

Note 1: The ZigBee PRO stack can later be reset to its default state (deleting context data except NWK frame counters) using the **zps_vDefaultStack()** function.

Note 2: The IEEE 802.15.4 MAC capabilities of a Router or End Device can be configured by the application using **zps_vAplAfSetMacCapability()** function.

5.1.1 Starting the Co-ordinator

The Co-ordinator must be the first node to be started. This node is pre-configured using the ZPS Configuration Editor. The functions that must be called in the Co-ordinator application to initialise the node are those listed at the start of this section (Section 5.1).

Once the stack has been started using **zps_eAplZdoStartStack()**, the Co-ordinator works through the following process to establish a network:

1. **Sets the radio channel for the network**

   The choice of 2.4-GHz band channel for the network is pre-configured via the the ZPS Configuration Editor (see Section 12.4.3) as either a fixed channel (in the range 11-26) or a set of channels from which the best channel will be selected by the Co-ordinator. In the latter case, the Co-ordinator performs an energy scan of the possible channels and chooses the quietest channel.

2. **Sets the Extended PAN ID for the network**

   The 64-bit Extended PAN ID (EPID) for the network is obtained as follows:

   - A pre-configured value may be set in the advanced device parameter *APS Use Extended PAN ID* in the ZPS Configuration Editor (see Section 12.4.4).
   - If the pre-set value is zero, the Co-ordinator will use its own IEEE/MAC address as the EPID.

   Note that the application may over-ride the EPID value set by the ZPS Configuration Editor by calling **zps_eAplAibSetApsUseExtendedPanId()** before calling **zps_eAplZdoStartStack()**.

3. **Accepts join requests from other devices (if enabled)**

   The Co-ordinator may now allow other devices (Routers and End Devices) to join the network as its children, enabling the network to grow. A maximum number of (direct) children of the Co-ordinator is pre-set via the advanced network parameters *Active Neighbour Table Size* and *Child Table Size* in the ZPS Configuration Editor (see Section 12.4.4), beyond which the Co-ordinator will not accept any further join requests from prospective children.

> **Note:** The initial 'permit joining' status is pre-set via the Co-ordinator parameter *Permit Joining Time* in the ZPS Configuration Editor. If this is initially disabled, the Co-ordinator may not accept children until joining has been enabled using **zps_eAplZdoPermitJoining()**. However, the 'permit joining' status is ignored during a join in which the pre-set EPID on the joining device is non-zero and during any rejoin (see Section 5.6.2). The above function can be used at any time to allow joinings for a limited time-period or indefinitely, and can also be used to disable joinings.

Once the Co-ordinator (and therefore network) has started, the stack event zps_EVENT_NWK_STARTED is generated on the device. If the Co-ordinator fails to start, the stack event zps_EVENT_NWK_FAILED_TO_START is generated.

When a node joins the Co-ordinator, the stack event zps_EVENT_NWK_NEW_NODE_HAS_JOINED is generated on the Co-ordinator.

5.1.2 Starting Routers and End Devices

A Router or End Device is pre-configured using the ZPS Configuration Editor. The functions that must be called in a Router or End Device application to initialise the node are those listed at the start of this section (Section 5.1).

> **Note:** The start-up and join process described in this section is for a first-time join (cold start) only and not for a rejoin (which is described in Section 5.6.2).

Once the stack has been started using **zps_eAplZdoStartStack()**, a Router or End Device works through the following process to join a network:

1. **Searches for a network to join**

   As part of the **zps_eAplZdoStartStack()** function call, the device searches for networks by listening for beacons from Routers and Co-ordinators of ZigBee PRO networks in the neighbourhood. The radio channel for this search is pre-configured via the ZPS Configuration Editor (see Section 12.4.3) in the same way as for the Co-ordinator as either a fixed channel (in the range 11-26) or a set of channels to scan. Thus, the device listens for beacons in the relevant channel(s). A beacon filter can be optionally introduced using the function **zps_bAppAddBeaconFilter()** to allow only beacons from networks of interest to be considered - beacons can be filtered on the basis of PAN ID, Extended PAN ID, LQI value and device joining status/capacity (see Appendix B.4).

   On completion of this search, the subsequent actions depend on the pre-set value of the 64-bit Extended PAN ID (EPID), which is set via the advanced device parameter *APS Use Extended PAN ID* in the ZPS Configuration Editor (see Section 12.4.4):

   - If the pre-set EPID value is non-zero, this value identifies a specific network to join (assuming the Co-ordinator has been pre-set with the same EPID - see Section 5.1.1). Provided that a network with this EPID has been discovered in the search, the device attempts to join this network as described in Step 3 below (therefore bypassing Step 2).
   - If the pre-set EPID value is zero, the results of the search are reported in a zps_EVENT_NWK_DISCOVERY_COMPLETE stack event, which contains details of the networks discovered (see Section 5.2.1). The device must then select a network to join, as described in Step 2 below.

2. **Selects a network to join**

   On the basis of the results in zps_EVENT_NWK_DISCOVERY_COMPLETE, the application must select a network which the device will attempt to join. The search results contain a recommended network, selected as the first ZigBee

PRO network detected that is allowing nodes to join. The application is, however, free to choose another network, where this choice may be based on LQI value (detected signal strength).

3. **Submits a join request to network**

   Once the device has identified a network to join, a request to join the network must be submitted. If a non-zero pre-configured EPID has been set (see above), this join request is submitted automatically, otherwise the function **zps_eAplZdoJoinNetwork()** must be called to submit the request. The outcome of this request is reported in one of the following stack events on the requesting device:

   - zps_EVENT_NWK_JOINED_AS_ROUTER (if joined as Router)
   - zps_EVENT_NWK_JOINED_AS_ENDDEVICE (if joined as End Device)
   - zps_EVENT_NWK_FAILED_TO_JOIN (if failed to join)

   In the case of success, the above stack event contains the 16-bit network address that the network has allocated to the local device. In addition, the event zps_EVENT_NWK_NEW_NODE_HAS_JOINED is generated on the parent.

   If the case of failure, the device can attempt another join by calling **zps_eAplZdoJoinNetwork()** with a different result reported in the zps_EVENT_NWK_DISCOVERY_COMPLETE event.

4. **Records the network's EPID for application use**

   The function **zps_eAplAibSetApsUseExtendedPanId()** may now be used to create a persistent record of the EPID of the network that the node has joined (it will first be necessary to obtain the EPID value using the functions **zps_pvAplZdoGetNwkHandle()** and **zps_u64NwkGetEpid()**). If this EPID record is created, the node will automatically continue in the network following a reset without explicitly rejoining.

5. **Router accepts join requests from other devices (if enabled)**

   A Router may now allow other devices (Routers and End Devices) to join it as its children. The number of (direct) children of the Router will be limited by the maximum number of neighbours for the node, which is pre-set via the advanced network parameter *Active Neighbour Table Size* and *Child Table Size* in the ZPS Configuration Editor (see Section 12.4.4).

> **Note:** The initial 'permit joining' status is pre-set via the Router parameter *Permit Joining Time* in the ZPS Configuration Editor. If this is initially disabled, the Router may not accept children until joining has been enabled using **zps_eAplZdoPermitJoining()**. However, the 'permit joining' status is ignored during a join in which the pre-set EPID on the joining device is non-zero and during any rejoin (see Section 5.6.2). The above function can be used at any time to allow joinings for a limited time-period or indefinitely, and can also be used to disable joinings.

Once a node has joined the network, each endpoint application on the node is next likely to search for compatible endpoints on remote nodes with which it can communicate, as described in Section 5.2.2.

> **Note:** A network can be set up such that an End Device or Router joins a particular parent node. The required configuration and function calls to employ pre-determined parents are described in Section 5.1.3.

5.1.3 Pre-determined Parents

It is possible to force a parent (Router or the Co-ordinator) to accept certain nodes as its (direct) children. The function **zps_eAplZdoDirectJoinNetwork()** can be used on this parent to register a potential child node (with specified IEEE/MAC and network addresses) by adding this node to the Neighbour table - *never write to the Neighbour table directly*. The parent then regards this node as an orphaned child. This function should only be called when the parent node is fully up and running - that is, the node has been started as described in Section 5.1.1 or Section 5.1.2.

When one of the designated children is started, its application should call the function **zps_eAplZdoOrphanRejoinNetwork()** in order to attempt to join the network as if it were a previously orphaned node. This function will start the ZigBee PRO stack and attempt to join the network whose EPID has been pre-configured on the node (using the ZPS Configuration Editor). The function will only allow the node to join a parent that already has knowledge of the node (in the parent's Neighbour table).

> **Note 1:** When **zps_eAplZdoOrphanRejoinNetwork()** is used, the start-up procedure described in Section 5.1.2 is not applicable to the joining node and the function **zps_eAplZdoStartStack()** must not be explicitly called on the node.
>
> **Note 2:** When a node joins the network in this way, the 'permit joining' status on the parent is ignored.

If the node successfully joins the network (via the designated parent), the stack event zps_EVENT_NWK_NEW_NODE_HAS_JOINED is generated on the parent node and one of the following stack events is generated on the joined node:

- zps_EVENT_NWK_JOINED_AS_ROUTER (if joined as a Router)
- zps_EVENT_NWK_JOINED_AS_ENDDEVICE (if joined as an End Device)

These events contain the network address that the parent has allocated to the joined node.

If the join request is unsuccessful, the zps_EVENT_NWK_FAILED_TO_JOIN event is generated on the joining node.

Once the node has joined the pre-determined parent, the node is next likely to search for compatible endpoints on remote nodes with which it can communicate, as described in Section 5.2.2.

5.2 Discovering the Network

This section describes how to discover properties of the network, including general network properties, node addresses and features, and the services offered by nodes. The important task of finding nodes that can communicate with each other is described. Maintenance of the 'primary discovery cache' of a node is also described - this cache contains information about other nodes of the network (not all nodes will host a primary discovery cache - only the Co-ordinator and Routers are allowed to).

5.2.1 Obtaining Network Properties

A 'network discovery' is implemented when the function **zps_eAplZdoStartStack()** is called to start the stack on an End Device or Router node (which needs to find a network to join). In addition, a network discovery can be explicitly started by calling the function **zps_eAplZdoDiscoverNetworks()**. For example, this function could be called if the initial network discovery did not find any suitable networks to join, in which case the function may be used to initiate a scan of previously unscanned channels (detailed in the stack event described below, resulting from the initial discovery).

Both of these function calls eventually result in the stack event zps_EVENT_NWK_DISCOVERY_COMPLETE on the End Device or Router, where this event reports the following properties of the discovered networks:

- Extended PAN ID
- ZigBee version
- ZigBee stack profile

This stack event also indicates the recommended network to join, which is taken to be the first ZigBee PRO network detected that is allowing nodes to join.

For information on joining a network, refer to Section 5.1.2.

5.2.2 Finding Compatible Endpoints

An endpoint on a newly joined node must find compatible endpoints on remote nodes with which to communicate. The decision of whether a remote endpoint is compatible is based on the endpoint properties stored in its Simple descriptor, notably the input/output clusters supported.

The endpoint application can discover compatible nodes by sending out a Match_Desc_req request identifying the required clusters. This request is submitted by calling the function **zps_eAplZdpMatchDescRequest()**, which allows the request to be sent as a broadcast to all nodes or as a unicast to a particular node (the sending node may already have a record of the network nodes and their addresses, as each node automatically announces itself in a broadcast when it joins the network). The request is sent in an APDU (Application Protocol Data Unit) which must first be allocated using the PDUM function **PDUM_hAPduAllocateAPduInstance()**.

A receiving endpoint which satisfies the supplied criteria replies to the request with a Match_Desc_rsp response which, when received, must be collected on the requesting node using the function **ZQ_bZQueueReceive()**. The requesting application may bind to a compatible endpoint (see Section 5.4) and communicate with the endpoint using binding or addressing (see Section 5.5).

5.2.3 Obtaining and Maintaining Node Addresses

The addresses of network nodes are needed in order to access node information (see Section 5.2.4), send data from one node to another (see Section 5.5) and bind nodes together (see Section 5.4). In most of these operations, an application can specify either 64-bit IEEE/MAC addresses or 16-bit network addresses, but the ZigBee PRO stack always works with network addresses. If the IEEE address (rather than the network address) of a remote node is specified by the application, the network address must still be available to the stack in an Address Map - see below.

The IEEE address of a node is assigned at the time of device manufacture and is fixed, while its network address is dynamically allocated by its parent when the device joins the network (this address may change if the network is re-started or the device later leaves and rejoins the network). Functions are provided to obtain the IEEE address of a node given its network address or to obtain the network address given the IEEE address. Use of these functions is described in Section 5.2.3.1 and Section 5.2.3.2.

> **Note:** The IEEE/MAC and network addresses of a node can be broadcast to all other nodes in the network using the function **zps_eAplZdpDeviceAnnceRequest()**. For example, this function would typically be called when the node joins or rejoins the network. The information is sent in a Device_annce announcement, which must be collected by the recipient nodes using the function **ZQ_bZQueueReceive()**.

An Address Map table can be maintained on a node, where each entry of this table contains the pair of addresses for a remote node - the 64-bit IEEE/MAC address and 16-bit network address. In fact, the IEEE/MAC address is not directly stored in the Address Map table but in a MAC Address table - the Address Map table contains the index of this address in the MAC Address table. The Address Map is automatically updated by the stack when a Device_annce announcement is received from a remote node (described in the Note above), but you can also add an address-pair to this table using the function **zps_eAplZdoAddAddrMapEntry()** - *never write to the Address Map table directly*. The Address Map must be properly maintained if the application employs IEEE/MAC addresses to identify remote nodes. In addition, when application-level security (see Section 5.8) is used in sending data from one node to another, the Address Map on the sending node must contain an entry for the target node.

5.2.3.1 Obtaining IEEE Address

You may wish to obtain the IEEE address of the node with a given network address - for example, in order to know which physical node corresponds to a particular dynamically allocated network address.

The IEEE address of the local node can be obtained simply by calling the function **zps_u64AplZdoGetIeeeAddr()**.

The IEEE address of a remote node can be obtained in either of two ways, depending on whether an entry for the node exists in the local Address Map table:

- The function **zps_u64AplZdoLookupIeeeAddr()** can be used to search the local Address Map table for the IEEE address which corresponds to a given network address.
- The required IEEE address can be obtained directly from the remote node by using the function **zps_eAplZdpIeeeAddrRequest()** to submit a request for the IEEE address of the node with a particular network address. This request, of type IEEE_addr_req, is sent in an APDU (Application Protocol Data Unit) which must first be allocated using the PDUM function **PDUM_hAPduAllocateAPduInstance()**. The request details are specified through the structure `zps_tsAplZdpIeeeAddrReq`, which includes an option to also request the IEEE addresses of all the target node's children (if any). The results are reported in an IEEE_addr_resp response.

5.2.3.2 Obtaining Network Address

You may wish to obtain the network address of the node with a given IEEE address - for example, in order to know the network address that has been dynamically allocated to a particular physical node.

The network address of the local node can be obtained simply by calling the function **zps_u16AplZdoGetNwkAddr()**.

The network address of a remote node can be obtained in either of two ways, depending on whether an entry for the node exists in the local Address Map table:

- **zps_u16AplZdoLookupAddr()** can be used to search the local Address Map table for the network address which corresponds to a given IEEE address.
- The required network address can be obtained directly from within the network by using the function **zps_eAplZdpNwkAddrRequest()** to submit a request for the network address of the node with a particular IEEE address. This request can be either unicast or broadcast, as follows:
  - Unicast to another node that will 'know' the required network address (this may be the parent of the node of interest or the Co-ordinator)
  - Broadcast to the network

  This request, of type NWK_addr_req, is sent in an APDU (Application Protocol Data Unit) which must first be allocated using the PDUM function **PDUM_hAPduAllocateAPduInstance()**. The request details are specified through the structure `zps_tsAplZdpNwkAddrReq`, which includes an option to also request the network addresses of all the target node's children (if any). The results are reported in a NWK_addr_resp response.

5.2.4 Obtaining Node Properties

Functions are provided to obtain information about the properties of network nodes. Much of this information is held on a node in special structures, referred to as descriptors. Five types of descriptor are used:

- Node descriptor
- Node Power descriptor
- Simple descriptor
- User descriptor
- Complex descriptor

In addition to the above, information can be obtained about the active endpoints, primary discovery cache and services of a node.

The required functions are detailed below. Functions are provided to obtain descriptors from the local node and from a remote node. When obtaining information from a remote node, the function sends a request in an APDU (Application Protocol Data Unit) which must first be allocated using the PDUM function **PDUM_hAPduAllocateAPduInstance()**. The results of the request are reported in a response which must be collected using the function **ZQ_bZQueueReceive()**.

> **Note 1:** When obtaining a descriptor of a remote node, the request can be submitted to the node itself or to another node which may hold the required descriptor in its primary discovery cache.
>
> **Note 2:** The structures that contain the descriptors (referenced below) are described in Section 7.2 and Section 8.2.1.
>
> **Note 3:** Where 64-bit IEEE/MAC addresses are used to identify remote nodes, the corresponding 16-bit network addresses must be available in the local Address Map - see Section 5.2.3.

Node Descriptor

The Node descriptor contains basic information about the node, such as its ZigBee node type and the radio frequency bands supported. The following functions can be used to obtain a Node descriptor:

- **zps_eAplAfGetNodeDescriptor()** obtains the Node descriptor of the local node. The result is stored in a structure of type `zps_tsAplAfNodeDescriptor`.
- **zps_eAplZdpNodeDescRequest()** requests the Node descriptor of a remote node. The result is stored in a structure of type `zps_tsAplZdpNodeDescriptor`.

Power Descriptor

The Node Power descriptor contains information about the node's supported power sources and present power source. The following functions can be used to obtain a Power descriptor:

- **zps_eAplAfGetNodePowerDescriptor()** obtains the Node Power descriptor of the local node. The result is stored in a structure of type `zps_tsAplAfNodePowerDescriptor`.
- **zps_eAplZdpPowerDescRequest()** requests the Node Power descriptor of a remote node. The result is stored in a structure of type `zps_tsAplZdpNodePowerDescriptor`.

Note that elements of the Node Power descriptor can be set on the local node using the ZPS Configuration Editor.

Simple Descriptor

There is a Simple descriptor for each endpoint on a node. The information in this descriptor includes the ZigBee device type supported by the endpoint as well as details of its input and output clusters. The following functions can be used to obtain a Simple descriptor:

- **zps_eAplAfGetSimpleDescriptor()** obtains the Simple descriptor of a particular endpoint on the local node. The result is stored in a structure of type `zps_tsAplAfSimpleDescriptor`.
- **zps_eAplZdpSimpleDescRequest()** requests the Simple descriptor of a particular endpoint on a remote node. The result is stored in a structure of type `zps_tsAplZdpSimpleDescReq`.

The returned Simple descriptor includes a list of input clusters and a list of output clusters of the endpoint.

When requesting a Simple descriptor from a remote node, if the cluster lists are long, the Simple descriptor may not fit into the APDU of the response. In this case, the returned Simple descriptor will contain incomplete cluster lists, but the remainder of the lists can be recovered using **zps_eAplZdpExtendedSimpleDescRequest()**.

It is also possible to search for nodes on the basis of certain criteria in the Simple descriptors of their endpoints - for example, search for endpoints which have a particular list of input clusters and/or output clusters. Such a search can be performed using the function **zps_eAplZdpMatchDescRequest()**. Use of this function is described in Section 5.2.2.

User Descriptor

The User descriptor is a user-defined character string, normally used to describe the node (e.g. “Thermostat”). The maximum length of the character string is 16, by default. A node need not have a User descriptor - if it has one, this must be indicated in the Node descriptor. The following functions can be used to access a User descriptor:

- **zps_eAplZdpUserDescSetRequest()** sets the User descriptor of a remote node.
- **zps_eAplZdpUserDescRequest()** requests the User descriptor of a remote node. The result is stored in a structure of type `zps_tsAplZdpUserDescReq`.

The above functions can only be used to access the User descriptor of a non-NXP device (which supports this descriptor), since the storage of a User descriptor on an NXP device is not supported.

Complex Descriptor

The Complex descriptor is an optional descriptor which contains device information such as manufacturer, model and serial number. The function **zps_eAplZdpComplexDescRequest()** allows the Complex descriptor of a remote node to be requested. However, the NXP ZigBee PRO stack does not support the functionality to produce a valid response and this function is provided only for compatibility with non-NXP products that do support the relevant functionality.

Active Endpoints

An endpoint on the local node can be configured as enabled or disabled using the function **zps_eAplAfSetEndpointState()**. An enabled endpoint is described as ‘active’. The current state of a local endpoint can be obtained using the function **zps_eAplAfGetEndpointState()**.

It is also possible to configure whether a local endpoint will be included in the results of network discovery operations, e.g. when **zps_eAplZdpMatchDescRequest()** is called. The ‘discoverable’ state of a local endpoint can be set using the function **zps_eAplAfSetEndpointDiscovery()**, while this state can be obtained using the function **zps_eAplAfGetEndpointDiscovery()**.

A list of the active endpoints on a remote can be obtain using the function **zps_eAplZdpActiveEpRequest()**. This functions submits an Active_EP_req request to the target node, which replies with an Active_EP_rsp response. If the active endpoint list is too long to fit into the APDU of the response, the returned list will be incomplete. However, the remainder of the list can be recovered using the function **zps_eAplZdpExtendedActiveEpRequest()**. Note that an endpoint is included in the list only if it is active and discoverable.

Primary Discovery Cache

A ZigBee routing node (Router or the Co-ordinator) may be able to host a 'primary discovery cache'. This is a database, held in memory, containing 'discovery information' about a set of network nodes, normally children and possibly other descendant nodes. The information held about a node includes the node's addresses, descriptors (Node, Node Power, Simple) and its list of active endpoints. Remote nodes can then interrogate the primary discovery cache to obtain information about other nodes in the network.

> **Note:** NXP nodes do not have the capability to hold a primary discovery cache, but functions are provided to interface with a primary discovery cache held on a node from another manufacturer.

The function **zps_eAplZdpDiscoveryCacheRequest()** allows nodes which hold a primary discovery cache to be detected. This function submits a Discovery_Cache_req request to the network. Nodes with a primary discovery cache reply with a Discovery_Cache_rsp response.

In addition, the function **zps_eAplZdpFindNodeCacheRequest()** can be used to search for nodes with a primary discovery cache that holds information about a particular node. This function submits a Find_node_cache_req request to the network. Nodes with the required node information in their caches reply with a Find_node_cache_rsp response.

Functions for storing node information in a primary discovery cache are described in Section 5.2.5.

Servers

A node can host one or more of the following 'servers' in a ZigBee PRO network:

- Primary Trust Centre
- Backup Trust Centre
- Primary Binding Table Cache
- Backup Binding Table Cache
- Primary Discovery Cache
- Backup Discovery Cache
- Network Manager

The function **zps_eAplZdpSystemServerDiscoveryRequest()** can be used to discover the servers hosted by other nodes in the network. The function broadcasts a System_Server_Discovery_req request to all nodes. A remote node replies with a System_Server_Discovery_rsp response containing a bitmap indicating the servers hosted by the node.

5.2.5 Maintaining a Primary Discovery Cache

Some routing nodes of a ZigBee PRO network may be capable of hosting a primary discovery cache, which contains 'discovery information' relating to other nodes in the network - see Primary Discovery Cache on page 78.

> **Note:** NXP nodes do not have the capability to hold a primary discovery cache, but functions are provided to interface with a primary discovery cache held on a node from another manufacturer.

Functions are provided for storing the local node's 'discovery information' in another node's primary discovery cache (normally in the parent or another ascendant node). First of all, **zps_eAplZdpDiscoveryStoreRequest()** must be called to allocate memory space for this information in the remote node's cache. This function sends a Discovery_store_req request to the remote node, which replies with a Discovery_store_rsp response. The local node's information can then be stored in the remote node's primary discovery cache using the following functions (which all operate on a request/response basis):

- **Node descriptor:** Stored using **zps_eAplZdpNodeDescStoreRequest()**
- **Power descriptor:** Stored using **zps_eAplZdpPowerDescStoreRequest()**
- **Simple descriptor:** Stored using **zps_eAplZdpSimpleDescStoreRequest()**
- **Active endpoints list:** Stored using **zps_eAplZdpActiveEpStoreRequest()**

A node's information can be removed from a primary discovery cache using the function **zps_eAplZdpRemoveNodeCacheRequest()**. This function can be called on the local node to remove a third node's information from the primary discovery cache of a remote node.

5.2.6 Discovering Routes

The route from one network node to another can be pre-established by implementing a route discovery. As a result, each routing node along the route will contain a Routing table entry for the destination node, where this entry consists of the destination address and the 'next hop' address. Routing and route discovery are fully introduced in Section 2.5.

Two functions are provided in the ZigBee PRO API to initiate route discoveries:

- **zps_eAplZdoRouteRequest()** can be used to establish a route from the local node to a specific destination node. This kind of end-to-end route discovery is outlined in Section 2.5.2.
- **zps_eAplZdoManyToOneRouteRequest()** can be used on a 'concentrator' node to implement a 'many-to-one' route discovery back to itself. The result is that Routing tables in routing nodes within a certain radius of the concentrator will acquire entries with the concentrator as the destination. Many-to-one routing is outlined in Section 2.5.3.

5.3 Managing Group Addresses

A 'group address' is a concept that simplifies data transfers (see Section 5.5) to multiple nodes/endpoints. It is a collective 16-bit address which refers to a group of destination endpoints (that may be located on different nodes). So, for example, when a group address is specified as the destination address for a data transfer, the data will be delivered to all the nodes/endpoints in the associated group. It is the responsibility of the wireless network application to allocate and manage group addresses on a network-wide basis.

A node which is to receive group-addressed communications must have a Group Address table. This table contains information about all the groups to which endpoints on the node belong - that is, each group address and the associated local endpoint numbers. The table is consulted on receiving a data packet with a group address - if the group address exists in the table, the packet is passed to the corresponding endpoint(s).

A Group Address table is created on a node using the ZPS Configuration Editor. The table can then be maintained by the application as follows:

- An endpoint can be added to a group by calling the function **zps_eAplZdoGroupEndpointAdd()** on the local node (which contains the endpoint).
- An endpoint can be removed from a group by calling the function **zps_eAplZdoGroupEndpointRemove()** on the local node (which contains the endpoint). Alternatively, **zps_eAplZdoGroupAllEndpointRemove()** can be used to remove a specified local endpoint from all groups to which it belongs.

The group addresses used in a network are defined by the application developer.

5.4 Binding

For the purpose of data communication between applications running on different nodes, it may be useful to 'bind' the relevant source and destination endpoints. When data is subsequently sent from the source endpoint, it is automatically routed to the bound destination endpoint(s) without the need to specify a destination address. For example, a binding could be created between the temperature sensor endpoint on a thermostat node and the switch endpoint on a heating controller node. Details of a binding are held in a Binding table on the source node. Binding is introduced more fully in Section 2.6.2, where bindings are one-to-one, one-to-many or many-to-one.

This section describes setting up a Bind Request Server and how to bind together two nodes, as well as how to unbind them. Access to the Binding tables is also described.

Note: Where 64-bit IEEE/MAC addresses are used to identify remote nodes, the corresponding 16-bit network addresses must be available in the local Address Map - see Section 5.2.3.

5.4.1 Setting Up Bind Request Server

A Bind Request Server must be set up on each device that will be the source node of a bound data transfer. This server manages a bound data transfer so that application processing is not blocked by concurrent requests for transmissions to the multiple destinations of the transfer. It does this by limiting the number of destinations and inserting a time delay between consecutive transmissions of a bound transfer.

The server is configured in the ZPS Configuration Editor (introduced in Chapter 12). Two parameter values must be set:

- ***Simultaneous Requests***

  This refers to the maximum number of destinations for a bound data transfer. The value set must be less than or equal to the value of the ZigBee network parameter *Maximum Number of Simultaneous Data Requests* or *Maximum Number of Simultaneous Data Requests with Acks*, described in Section 11.7.

- ***Time Interval***

  This refers to the time interval between consecutive transmissions to the different destinations of a bound data transfer and is measured in milliseconds.

In the ZPS Configuration Editor, these parameters are accessed by clicking on **Bind Request Server** under **ZDO Configuration** for the device (the parameters appear in the **Properties** tab of the bottom pane).

> **Note:** The bound server can only handle one bound request at a time. The application must wait for the confirmation from the first bound request before attempting to send a second bound request.

5.4.2 Binding Endpoints

An endpoint on the local node can be bound to one or more endpoints on remote nodes using the following functions:

- **zps_eAplZdoBind()** creates a one-to-one binding to a single remote endpoint.
- **zps_eAplZdoBindGroup()** creates a one-to-many binding for which the destination endpoints are specified via a group address (refer to Section 5.3).

The function **zps_eAplZdpEndDeviceBindRequest()** is also provided, which allows an endpoint on one End Device to be bound to an endpoint on another End Device via the Co-ordinator. This function must be called on both End Devices, where the function call would typically be triggered by a user action such as pressing a button on the node. The function submits an End_Device_Bind_req request to the Co-ordinator, which replies with an End_Device_Bind_rsp response. The stack will then automatically update the Binding tables on the End Devices (as the result of bind requests from the Co-ordinator), and these updates will be indicated by a zps_EVENT_ZDO_BIND event on each of the End Devices.

5.4.3 Unbinding Endpoints

Bindings can be removed using the following functions:

- Two endpoints previously bound using **zps_eAplZdoBind()** can be unbound using the function **zps_eAplZdoUnbind()**.
- Endpoints previously bound using **zps_eAplZdoBindGroup()** can be unbound using the function **zps_eAplZdoUnbindGroup()**.

5.4.4 Accessing Binding Tables

Information about established bindings is held in Binding tables on the relevant nodes. Normally, a Binding table is held on a node which contains at least one source endpoint for a binding - thus, the table includes entries for all bindings which involve source endpoints on the local node. Alternatively, the Binding table entries for a particular source node can be held in a primary Binding table cache on the node's parent or another ascendant node. However, if a primary Binding table cache exists on an ascendant node, a source node can opt out of membership of this table by calling the function **zps_eAplZdpBindRegisterRequest()** to indicate that the source node will store its own Binding table entries locally.

Functions are provided which allow Binding tables to be remotely accessed and modified. These functions are particularly useful in implementing a commissioning tool application.

A binding can be remotely created or removed by requesting a modification to the relevant Binding table on a remote node. The remote Binding table may be a primary Binding table cache or the source node's local Binding table, whichever is relevant for the particular binding.

- The function **zps_eAplZdpBindUnbindRequest()** can be used to request that a new binding is added to a remote Binding table. The addition of this binding is signalled by a zps_EVENT_ZDO_BIND event on the remote node.
- The function **zps_eAplZdpBindUnbindRequest()** can also be used to request that an existing binding is removed from a remote Binding table. The removal of this binding is signalled by a zps_EVENT_ZDO_UNBIND event on the remote node. A Binding table entry can also be removed locally using the function **zps_eAplAibRemoveBindTableEntryForMacAddress()**, which requests that the entry containing a particular IEEE/MAC address is deleted.

In addition, binding entries in a remote primary Binding table cache can be modified using the function **zps_eAplZdpReplaceDeviceRequest()**, to replace an IEEE/MAC address and/or endpoint number. This operation works on a 'search and replace' basis in the Binding table, and the address/endpoint number to be replaced could occur in the source or destination of one or more table entries.

The function **zps_eAplZdpMgmtBindRequest()** is also provided, which can be used to request the Binding table of a remote node.

5.5 Transferring Data

This section describes how to send data to a remote node and receive the data at the destination. The data polling method is also described, which is used by an End Device to obtain data that arrives at its parent while the End Device is asleep.

5.5.1 Sending Data

Data is sent across the wireless network in an Application Protocol Data Unit (APDU). Before calling the function to send the data, an APDU instance must first be allocated using the PDUM function **PDUM_hAPduAllocateAPduInstance()** and then populated with data using the PDUM function **PDUM_u16APduInstanceWriteNBO()**.

There are five ways to send data to one or more remote nodes:

- **Unicast:** Sending data to a single destination endpoint
- **Broadcast:** Sending data to (potentially) all endpoints
- **Group Multicast:** Sending data to a group of endpoints
- **Bound Transfer:** Sending data to bound endpoints
- **Inter-PAN Transfer:** Sending data to another ZigBee PRO network

These methods are described in the sub-sections below. However, in all cases except the inter-PAN transfer, a general function **zps_eAplAfApsdeDataReq()** can be used which imposes no restrictions on the destination address, destination cluster and destination endpoint number - these destination parameters do not need to be known to the stack or defined in the ZPS configuration.

> **Note 1:** In all cases, once the data packet has been successfully sent, a 'DATA_CONFIRM' stack event is generated. When sending data to one or more individual nodes (not broadcasting), this event is generated after a MAC-level acknowledgement has been received from the 'next hop' node.
>
> **Note 2:** Where 64-bit IEEE/MAC addresses are used to identify remote nodes, the corresponding 16-bit network addresses must be available in the local Address Map - see Section 5.2.3.

5.5.1.1 Unicast

A unicast is a data transmission to a single destination - in this case, a single endpoint. The destination node for a unicast can be specified using the network address or the IEEE/MAC address of the node:

- **zps_eAplAfUnicastDataReq()** is used to send a data packet to an endpoint on a node with a given network address.
- **zps_eAplAfUnicastIeeeDataReq()** is used to send a data packet to an endpoint on a node with a given IEEE/MAC address.

Neither of these functions provide any indication that the data packet has been successfully delivered to its destination. It is possible that a unicast packet will not reach its destination because the packet is lost - for example, it becomes caught in a circular route. However, equivalent functions are available which request the destination node to provide an acknowledgement of data received - these 'with acknowledgement' functions are **zps_eAplAfUnicastAckDataReq()** and **zps_eAplAfUnicastIeeeAckDataReq()**, requiring network and IEEE/MAC addresses respectively. These functions request end-to-end acknowledgements which, when received, generate zps_EVENT_APS_DATA_ACK events (note that the 'next hop' zps_EVENT_APS_DATA_CONFIRM events will also be generated). A timeout of approximately 1600 ms is applied to the acknowledgements. If an acknowledgement has not been received within the timeout period, the data is re-sent, and up to 3 more re-tries can subsequently be performed before the data transfer is abandoned completely (which occurs approximately 3 seconds after the initial send).

Note: If a message is unicast to a destination for which a route has not already been established, the message will not be sent and a route discovery will be performed instead. If this is the case, the unicast function will return zps_NWK_ENUM_ROUTE_ERROR. The application must then wait for the stack event zps_EVENT_NWK_ROUTE_DISCOVERY_CONFIRM (success or failure) before attempting to re-send the message by calling the same unicast function again.

Unicasts from Sleepy Nodes

To allow a unicast acknowledgment to be received as described above, the source node must remain awake for a time equal to the timeout period. On a battery-powered node which sleeps, the use of acknowledgements and retries may not be desirable from a power-saving point of view. In this case, acknowledgements should not be used, but it is good practice for the application to monitor the route to a remote node by periodically attempting to read an attribute on the node and wait for a response. If the response is not observed within a pre-defined time then the application should take one of the actions listed below, depending on whether the source node is an End Device or Router.

- If an End Device, the application should notify the parent node about the routing problem by sending it a unicast network status command using the function **zps_vNwkSendNwkStatusCommand()**, with the status as "No Route Available (0x00)"
- If a Router, the application should initiate an explicit route discovery to the destination node by calling the function **zps_eAplZdoRouteRequest()**

Fragmenting Large Unicast Packets

The unicast 'with acknowledgement' functions, **zps_eAplAfUnicastAckDataReq()** and **zps_eAplAfUnicastIeeeAckDataReq()**, also allow a large data packet to be sent that may be fragmented into multiple messages during transmission. Application design issues concerned with fragmented data transfers are outlined in Appendix B.1.

5.5.1.2 Broadcast

A broadcast is a data transmission to all network nodes, although it is possible to select a subset of nodes. The following destinations are possible:

- All nodes
- All nodes for which 'receiver on when idle' - these include the Co-ordinator, Routers and non-sleeping End Devices
- All Routers and the Co-ordinator

The function **zps_eAplAfBroadcastDataReq()** is used to broadcast a data packet. It is possible to specify a particular destination endpoint on the nodes (the same endpoint number for all recipient nodes) or all endpoints. Following this function call, the packet may be broadcast up to four times (in addition, the packet may be subsequently re-broadcast up to four times by each intermediate routing node).

5.5.1.3 Group Multicast

A group multicast is a data transmission which is intended for a selection of network nodes or, more specifically, a selection of endpoints on these nodes. The set of destination endpoints must be pre-assembled into a group with an associated 'group address', as described in Section 5.3.

The function **zps_eAplAfGroupDataReq()** is used to send a data packet to the group of endpoints with a given group address. In practice, the data packet is broadcast to all nodes in the network and it is the responsibility of each recipient node to determine whether it has endpoints in the target group (and therefore whether the packet is of interest).

5.5.1.4 Bound Transfer

A data packet can be sent from an endpoint to all the remote endpoints with which the source endpoint has been previously bound (see Section 5.4). The function **zps_eAplAfBoundDataReq()** is used to implement this type of data transfer. This method provides an alternative to a group multicast (see Section 5.5.1.3) for sending data to selected endpoints.

An equivalent to the above function is provided which also requests an 'end-to-end' acknowledgement from the destination - **zps_eAplAfBoundAckDataReq()**. If an acknowledgement has not been received within approximately 1600 ms of the initial request, the data is re-sent, with up to 3 more subsequent re-tries before the data transfer is abandoned completely.

zps_eAplAfBoundAckDataReq() also allows a large data packet to be sent that may need to be fragmented into multiple messages during transmission. Application design issues concerned with fragmented data transfers are outlined in Appendix B.1.

Following a call to one of the above bound transfer functions, a deferred zps_EVENT_BIND_REQUEST_SERVER event is generated on the sending node. This event summarises the status of the transmission (see Section 7.2.2.21), including the number of bound endpoints for which the transmission failed. The event is generated only after receiving MAC-level acknowledgments from the 'next hop' nodes or, if requested, after receiving end-to-end acknowledgments from the destination nodes.

Note: In the case of a bound transfer, the 'next hop' zps_EVENT_APS_DATA_CONFIRM events and 'end-to-end' zps_EVENT_APS_DATA_ACK events are consumed and do not reach the application.

5.5.1.5 Inter-PAN Transfer

A data packet can be sent to nodes in other IEEE 802.15.4 networks - this is referred to as an inter-PAN transfer or transmission. Typically, this mechanism could be used to send information to optional low-cost devices that are not part of the local network. Note that no security (encyption/decryption) can be applied to inter-PAN transfers and only one application on a device can perform inter-PAN transmissions. The inter-PAN messages are not forwarded and so will only be received by nodes within direct radio range of the transmitter.

The inter-PAN feature is enabled via the ZPS Configuration Editor. The *Inter PAN* value is set to true in the APS Layer Configuration section of the Advanced Properties for the device.

The function **zps_eAplAfInterPanDataReq()** is used to request an inter-PAN transmission. This function requires the destination(s) for the transfer to specified:

- Single destination node in a specific network
  (PAN ID and node address must be specified)
- Multiple destination nodes in a specific network
  (PAN ID and a group address for the nodes must be specified)
- All nodes in a specific network
  (PAN ID and broadcast address of 0xFFFF must be specified)
- All nodes in all reachable networks
  (broadcast PAN ID and broadcast address, both of 0xFFFF, must be specified)

After successfully sending the data packet, the stack will generate the event zps_EVENT_APS_INTERPAN_DATA_CONFIRM (for a single destination, this event is generated once the ‘next hop’ acknowledgement has been received).

A destination endpoint is not specified for this type of data transfer but a cluster must be specified for the destination. On receiving the data packet, the recipient node will automatically pass the packet to the endpoint which supports the given cluster (see Section 5.5.2).

5.5.2 Receiving Data

When a data packet (sent using one of the methods described in Section 5.5.1) is received by the destination node, it is put into a message queue. A zps_EVENT_AF_DATA_INDICATION stack event is generated on the destination node to indicate that a data packet has arrived (the destination endpoint is indicated in this event). The packet must then be collected from the message queue using the function **ZQ_bZQueueReceive()**.

> **Note 1:** In the case of a data packet received from another network by means of an inter-PAN transfer, the zps_EVENT_APS_INTERPAN_DATA_INDICATION stack event will be generated. The data packet will be passed to the endpoint which supports the specified cluster. The application must always handle these inter-PAN packets and release the APDU instances (see below). The event will only be generated if the inter-PAN feature has been enabled via the ZPS Configuration Editor. If an application transmits inter-PAN messages but does not need to receive them, the application must enable inter-PAN in the ZPS Configuration Editor and handle any zps_EVENT_APS_INTERPAN_DATA_INDICATION events by releasing the APDU instances.
>
> **Note 2:** In the case of the arrival of a response packet which is destined for the ZDO, a zps_EVENT_AF_DATA_INDICATION stack event will be generated with a destination endpoint of 0. It will be necessary for the application to call the function **zps_bAplZdpUnpackResponse()** to extract the response data from the event.

An End Device which is asleep will be unable to receive a data packet directly, so the data is buffered by its parent for collection later. The End Device must explicitly request this data, once awake. This method of receiving data is called data polling and is described in Section 5.5.3.

Once a data packet has been collected from a message queue, the data can be extracted from the APDU instance using the PDUM function **PDUM_u16APduInstanceReadNBO()**. The APDU instance must then be released using the PDUM function **PDUM_eAPduFreeAPduInstance()**.

5.5.3 Polling for Data

In the case of an End Device which is capable of sleeping, messages are not delivered directly to the device, since it may be asleep when the messages arrive. Instead, the messages are temporarily buffered by the End Device's parent. Once awake, the End Device can then ask or 'poll' its parent for data.

> **Note:** End Devices that are not enabled for sleep can receive messages directly and therefore do not need to poll. An End Device is pre-configured as either sleeping or non-sleeping via the End Device parameter *Sleeping* in the ZPS Configuration Editor (see Section 12.4.2).

Data polling is performed using the function **zps_eAplZdoPoll()** in the End Device application. This function requests the buffered data and should normally be called immediately after waking from sleep. If the poll request is successfully sent to the parent, a zps_EVENT_NWK_POLL_CONFIRM stack event will occur on the End Device. The subsequent arrival of data from the parent is indicated by the stack event zps_EVENT_AF_DATA_INDICATION. Any messages forwarded from the parent should then be collected from the relevant message queue using the function **ZQ_bZQueueReceive()**, just as for normal data reception described in Section 5.5.2.

Application design issues concerned with transferring data to a sleeping End Device are outlined in Appendix B.2.

5.5.4 Security in Data Transfers

The 'send data' functions for unicast, broadcast, group transfer and bound transfer contain a parameter to select the required security setting for the protection of the sent message. In the NXP ZigBee PRO software, there are currently three security options, as follows:

- No security
- Network-level security
- Application-level security

Application-level security is only available for unicast and bound transfers, while network-level security is available for all transfer types except inter-PAN transfers.

Network-level and application-level security are detailed in Section 5.8.

> **Note 1:** No security is available for inter-PAN transfers (to other networks).
>
> **Note 2:** When application-level security is used in sending data, the IEEE/MAC address and network address of the target node must be available through the local Address Map table - see Section 5.2.3.

5.6 Leaving and Rejoining the Network

This section describes how a node may leave the network and later rejoin either the same network or a different network.

5.6.1 Leaving the Network

A node may leave the network intentionally or unintentionally:

- The node may be intentionally (and temporarily) removed from the network for maintenance work, such as the replacement of batteries.
- The node may unintentionally leave the network due to unforeseen circumstances, such as a broken radio link with its parent (an obstacle may have been introduced into the path of the signal).

A node can be intentionally removed from the network using the function **zps_eAplZdoLeaveNetwork()**, which issues a leave request. The target node can be the requesting node itself or a child of the requesting node. The application may be designed to call this function when a button is pressed on the requesting node.

When calling **zps_eAplZdoLeaveNetwork()**:

- You can specify whether the children of the leaving node should also be requested to leave the network. If this is the case, the leaving node will first automatically call **zps_eAplZdoLeaveNetwork()** for each of its children.
- You can specify whether the leaving node should immediately attempt to rejoin the same network after leaving (see Section 5.6.2).

The stack event zps_EVENT_NWK_LEAVE_INDICATION is generated on the node which has been requested to leave (this event is also generated when a neighbouring node has left the network). Once a node has been successfully removed from the network as the result of a call to **zps_eAplZdoLeaveNetwork()**, the stack event zps_EVENT_NWK_LEAVE_CONFIRM is generated on the requesting node.

The function **zps_eAplZdpMgmtLeaveRequest()** is also provided which can be used to request a remote node to leave the network.

By default, a Router will always act on leave request messages. However, it may be desirable for a Router to ignore leave request messages in order to prevent a rogue node from disrupting the network. If the function **zps_vNwkNibSetLeaveAllowed()** is called with the *bLeave* parameter as FALSE, the Router will ignore network leave requests. End Devices always act on leave requests from their parent and ignore leave requests from other nodes.

Alternatively, a callback function can be defined that is invoked when a leave request is received, where this function determines whether the leave request is to be obeyed - this decision may depend on where the leave request came from. The callback function is registered using **zps_eAplZdoRegisterZdoLeaveActionCallback()** - refer to the description of this function on page 94 for details of the callback function.

5.6.2 Rejoining the Network

A node may leave its network - for example, by:

- losing radio contact with its parent - the stack on the ‘orphaned’ node will detect this loss and automatically attempt to rejoin the network
- calling **zps_eAplZdoLeaveNetwork()** - the node will automatically attempt to rejoin the network if an immediate rejoin has been requested in the function call (although the node can be configured to always rejoin the network following a leave, using the function **zps_vNwkNibSetLeaveRejoin()**)

If the node successfully rejoins the network, the stack event zps_EVENT_NWK_NEW_NODE_HAS_JOINED is generated on the parent node and one of the following stack events is generated on the joined node:

- zps_EVENT_NWK_JOINED_AS_ROUTER (if joined as a Router)
- zps_EVENT_NWK_JOINED_AS_ENDDEVICE (if joined as an End Device)

These events contain the network address that the parent has allocated to the joined node (this may be different from the network address that the node previously had).

If the join request is unsuccessful, the zps_EVENT_NWK_FAILED_TO_JOIN event is generated on the requesting node.

If an automatic rejoin has failed or has not been requested, the function **zps_eAplZdoRejoinNetwork()** can be used to request a rejoin (this function must be called on the node that needs to rejoin). The application may be designed to call this function when a button is pressed on the node. The result of this function call will be indicated by means of the above events.

The function **zps_eAplZdpMgmtDirectJoinRequest()** is also provided which submits a request to a remote parent to allow a particular node to join it. In addition, the function **zps_eAplZdpMgmtPermitJoiningRequest()** is provided which allows joining to be enabled/disabled on a remote node.

Note 1: When a device rejoins a network, the ‘permit joining’ status on the potential parent is ignored.

Note 2: When a device joins the network, its application may call **zps_eAplZdpDeviceAnnceRequest()** to announce the device’s membership and network address to the rest of the network. The information is sent in a Device_annce announcement, which must be collected by the recipient nodes using the function **ZQ_bZQueueReceive()**.

> ***Caution:*** *If a node rejoins the same secured network but its stack context data was cleared before the rejoin (by calling **NvErase()**), data sent by the node will be rejected by the destination node since the frame counter has been reset on the source node. Therefore, you are not recommended to clear the stack context data before a rejoin. For more information and advice, refer to Appendix B.3.*

5.7 Return Codes and Extended Error Handling

When a ZigBee PRO API function is called, a code is normally returned on completion of the function to indicate the outcome. This code is taken from one of the following:

- zps_E_SUCCESS
- APS return codes, listed and described in Section 10.2.2
- NWK return codes, listed and described in Section 10.2.3
- MAC return codes, listed and described in Section 10.2.4

An extended error handling mechanism can be optionally implemented which allows more detail to be obtained about certain errors that can occur during function execution. The particular errors are:

- 0xA3: zps_APL_APS_E_ILLEGAL_REQUEST
- 0xA6: zps_APL_APS_E_INVALID_PARAMETER
- 0xC2: zps_NWK_ENUM_INVALID_REQUEST

The extended error codes are listed and described in Section 10.2.5.

In order to implement the extended error handling mechanism, you must register a callback function using the function **zps_vExtendedStatusSetCallback()**. This registration function must be called before invoking the first API function for which extended error handling is required. The registered callback function will then be invoked during execution of the API function if one of the above errors occurs. The callback function will return an extended error code (from those listed in Section 10.2.5) but the API function will return only the basic error code.

5.8 Implementing ZigBee Security

The NXP ZigBee PRO APIs allow ZigBee security to be implemented, which applies key-based encryption to communications between network nodes. The message frame content generated at the NWK layer and higher is encrypted using 128-bit AES-based encryption (see Section 1.8). The NWK payload of the frame is encrypted, and the NWK header and payload are integrity-protected with a 32-bit Message Integrity Code (MIC).

This section describes security in a network with centralised security that is managed by a single Trust Centre, which is usually the Co-ordinator node. A distributed security scheme can alternatively be used and this is described in Section 5.10.2.

The sub-sections below deal with the following topics:

- Security levels - see Section 5.8.1
- Security keys - see Section 5.8.2
- Security set-up - see Section 5.8.3
- Security key modification - see Section 5.8.4

5.8.1 Security Levels

Two types or levels of security can be applied in a ZigBee network:

- **Network-level security**: This uses a ‘network key’ which is common throughout the network and is used to encrypt/decrypt all communications between all nodes. The network key is randomly generated by the Trust Centre before any nodes join the network. Setting up network-level security is described in Section 5.8.3.1.
- **Application-level security**: This uses an application ‘link key’ which is used (in addition to the network key) to encrypt/decrypt communications between a pair of nodes. This link key may be unique for a pair of nodes. Setting up application-level security is described in Section 5.8.3.2.

The encryption keys for these security levels are described in Section 5.8.2.

5.8.2 Security Key Types

The different ZigBee security keys are summarised in Table 4 on page 95 and described in more detail below.

When a node joins the network, the Trust Centre must pass the network key to the joining node, so that the node can participate in network-level encrypted communications with existing network nodes. The network key must itself be protected by encryption when it is passed to the joining node. For this encryption, a pre-configured link key is used, which is known by both the Trust Centre and the joining node. This can be a global link key or a unique link key:

- **Pre-configured global link key**: This link key is the same for all nodes in the network. It may be ZigBee-defined key or manufacturer-defined:
  - The ZigBee-defined key (known as the ZigBee “09” key) will allow nodes from different manufacturers to join the network.
  - A manufacturer-defined key will allow only nodes from the specific manufacturer to join the network.
- **Pre-configured unique link key**: This link key is an exclusive key for the Trust Centre and joining node. In this case, every node will have a different link key.

The pre-configured link key must be pre-programmed into the relevant nodes either in the factory or during commissioning.

The network-level security set-up process is described in Section 5.8.3.1. The active network key can subsequently be changed at any time, as described in Section 5.8.4.

Once network-level security is set up, application-level security can be set up for more secure communications - this level of security is applied on top of network-level security. If application-level security is required between two nodes then a link key must be established for the nodes. This key can be any of the following:

- **Pre-configured global link key** (as detailed above): This is for communications between the Trust Centre and all other nodes
- **Pre-configured unique link key** (as detailed above): This is for communications between the Trust Centre and one other node
- **Trust Centre Link Key (TCLK)**: This is for communications between the Trust Centre and one other node. It is randomly generated by the Trust Centre and passed to the relevant node, for which it is encrypted with the network key and, if it exists, the pre-configured unique link key for the node. The TCLK will then be used to encrypt all subsequent communications with the Trust Centre, replacing any pre-configured link key (the application should hold on to the pre-configured key in case it needs to be reinstated in the future, e.g. for a re-join).
- **Application link key:** This is for communications between a pair of nodes that does not include the Trust Centre. It is requested from the Trust Centre by one of the two nodes. The Trust Centre randomly generates the key and associates it with the IEEE/MAC addresses of the two nodes. The Trust Centre passes the key to each node, for which it is encrypted with the network key and, if it exists, the pre-configured unique link key for the node.

The application-level security set-up process is described in Section 5.8.3.2.

Security Key	Description	
Network-level Security		
Network key	• Essential key used to encrypt communications between all nodes of the network • Randomly generated by the Trust Centre • Distributed to joining nodes, encrypted with a pre-configured link key (see below)	
Application-level Security		
Global link key (pre-configured)	• Used between the Trust Centre and all other nodes • Pre-configured in all nodes (unless a unique link key is pre-configured - see below) • Also used in joining to encrypt network key transported from Trust Centre to joining node • If ZigBee-defined, allows nodes from all manufacturers to join the network • If manufacturer-defined, allows only nodes from one manufacturer to join the network • Touchlink Pre-configured Link Key is a key of this type • Distributed Security Global Link Key is a key of this type	
Unique link key	Optional key used to encrypt communications between a pair of nodes - may be one of:	
	Pre-configured unique link key	• Used between the Trust Centre and one other node • Pre-configured in Trust Centre and relevant node • Also used in joining to encrypt network key transported from Trust Centre to joining node • Install Code-derived Pre-configured Link Key is a key of this type
	Trust Centre Link Key (TCLK)	• Used between the Trust Centre and one other node • Randomly generated by the Trust Centre • Distributed to node encrypted with network key and pre-configured link key (if any) • Replaces pre-configured link key (if any) but application must retain the pre-configured key in case it needs to be reinstated
	Application link key	• Used between a pair of nodes, not including the Trust Centre • Randomly generated by the Trust Centre • Distributed to each node encrypted with network key and pre-configured link key (if any)

Table 4: ZigBee Security Key Summary

Note 1: A pre-configured unique link key for a node can be derived from an install code on the Trust Centre using the **zps_eAplZdoAddReplaceInstallCodes()** function. Install codes are described in the *ZigBee 3.0 Devices User Guide (JN-UG-3131)*.

Note 2: In order to use a pre-configured link key in a ZigBee 3.0 application that uses the ZigBee Base Device (see Section 2.4.4), the Base Device attribute `bbdbJoinUsesInstallCodeKey` must be enabled and set to TRUE - for more information, refer to the *ZigBee 3.0 Devices User Guide (JN-UG-3131)*.

5.8.3 Setting Up ZigBee Security

This section describes how to set up ZigBee security in your application code. Note that if security is enabled in a ZigBee network then network-level security is always used, while application-level security is optional.

Security is enabled on a node via the device parameter *Security Enabled* in the ZPS Configuration Editor. Enabling security also enables many-to-one routing towards the Trust Centre, which becomes a network concentrator (see Section 2.5.3).

A Trust Centre must be nominated (see Section 1.8) using the ZPS Configuration Editor. The Co-ordinator is normally chosen as the Trust Centre. The maximum number of nodes that will require the services of the Trust Centre must be set on the nominated node using the network parameter *Route Record Table Size* in the ZPS Configuration Editor (the default number is 4).

Security can be set up in the application code using the function **zps_vAplSecSetInitialSecurityState()**, which must be called before **zps_eAplAfInit()** and **zps_eAplZdoStartStack()** - see Section 5.1.

> **Note:** As an alternative to using the function **zps_vAplSecSetInitialSecurityState()** in the application code, ZigBee security can be set up in the ZPS Configuration Editor (see Section 5.8.3.1).

Once **zps_vAplSecSetInitialSecurityState()** has been called and the stack has been started, the stack will automatically manage the subsequent network-level security set-up and implementation.

Network-level security set-up and application-level security set-up are further described in Section 5.8.3.1 and Section 5.8.3.2 respectively.

> **Note:** Certain functionality on the Trust Centre can be disabled using the **zps_vSetTCLockDownOverride()** function. For more information, refer to the function description on page 109.

5.8.3.1 Network-level Security Set-up

The function **zps_vAplSecSetInitialSecurityState()**, described above, initiates the set-up process for network-level security and requires the type of initial security key to be specified as one of:

- Pre-configured global link key
- Pre-configured unique link key

These keys are described in Section 5.8.2. They are used to encrypt the network key when it is transported to a joining node.

The Trust Centre and other nodes must be pre-programmed with the relevant pre-configured link key(s). This key can be specified in the application code for the node and referenced by **zps_vAplSecSetInitialSecurityState()** or can be set through the Key Descriptor parameter *Key* in the ZPS Configuration Editor on both the Trust Centre and other node(s). In the case of a unique link key, the IEEE/MAC address of the node must also be pre-programmed into the Trust Centre along with the link key. For the Key Descriptor parameters, refer to Section 11.7.9.

Note: Pre-configured link keys entered via the ZPS Configuration Editor are held in a Key Descriptor Table on the Trust Centre, with one entry for each node/key. The key for a node with a given IEEE/MAC address can be obtained (locally) from this table using the function **zps_psGetActiveKey()**.

The Trust Centre generates a random network key to be used in network-level communications between all nodes. When a new node joins the network, the Trust Centre transports this network key, encrypted using the appropriate pre-configured link key, to the newly joined node.

Note 1: The application on the Trust Centre can take control (from the stack) of whether a node is allowed to join the network (possibly using its pre-configured link key) through a user-defined callback function. If required, this callback function must be registered using the function **zps_vTCSetCallback()**. For more details, refer to the function description on page 111.

Note 2: When a device joins a ZigBee network and requires authentication which involves transporting a network key to it, the parent opens an authentication interval during which the joining device must announce itself to the network. This interval begins from the transmission of a rejoin response (if the device joins through a NWK layer rejoin) or an association response (if it joins through an IEEE 802.15.4 association). If the device fails to announce itself during this interval, the parent removes the Neighbour table entry for the joining device to ensure that the child capacity of the parent is maintained. This authentication interval must be set on all potential parent nodes via the network parameter *APS Security Timeout Period* (see Section 11.7), which is 1 second by default but 6 seconds is a more reasonable setting.

5.8.3.2 Application-level Security Set-up

Once network-level security has been set up (as described in Section 5.8.3.1), application-level security can be set up, if required. In application-level security, the communications between two nodes are encrypted/decrypted using a link key which may be global or unique:

- **Global link key:** This is shared between all nodes on the network and is pre-configured in all the nodes. Frame counters are not checked for freshness when using a global link key.
- **Unique link key:** This is exclusive to a pair of nodes that need to communicate privately. It may be a pre-configured unique link key, Trust Centre Link Key (TCLK) or application link key. Frame counters are checked for freshness to prevent rogue nodes replaying stale messages. This provides the most secure method of application security.

The different types of link key are described in Section 5.8.2 and summarised in Table 4 on page 95.

In order to set up application-level security with a unique link key between two nodes, the function **zps_eAplZdoRequestKeyReq()** must be called on one of the nodes to request a link key from the Trust Centre. There are two possibilities:

- To request a Trust Centre Link Key (TCLK) for communication between the local node and the Trust Centre - the Trust Centre will respond with the requested link key
- To request an application link key for communication with another node that is not the Trust Centre (in this case, the IEEE/MAC address of the other node must be supplied in the function call) - the Trust Centre will send the requested link key to both nodes

The Trust Centre will ignore the request if the node is not permitted to send APS secured data. The Trust Centre responses are encrypted as follows:

- If a link key exists for communications between the Trust Centre and the target node (e.g. a pre-configured link key), this key and the network key are both used to encrypt the requested link key.
- Otherwise, only the network key is used to encrypt the requested link key.

On receiving the link key, the ZigBee stack on the node will automatically save the key. The event zps_EVENT_ZDO_LINK_KEY is generated to indicate that the link key is available. Any subsequent unicast or bound data transfer between these two nodes can opt to use this key (zps_E_APL_AF_SECURE_APL).

> **Note:** An application link key can be introduced directly by the application using the function **zps_eAplZdoAddReplaceLinkKey()**.

Note 1: When a link key is used to encrypt a data packet, the packet payload is encrypted at the application level using the link key and then the packet is encrypted at the ZigBee stack NWK layer using the network key (therefore, both keys are used).

Note 2: When application-level security is used in sending data, the IEEE/MAC address and network address of the target node must be available through the local Address Map table - see Section 5.2.3.

5.8.4 Security Key Modification

The network key and an application link key can be changed while the network is operating, as described below.

5.8.4.1 Network Key Modification

It is possible to store more than one network key on a node, although only one key can be active at any one time. Each network key is identified by means of a unique 'key sequence number' assigned by the Trust Centre application.

A new network key can be installed in a node in one of two ways:

- Distributed by the Trust Centre to one or multiple nodes of the network using the function **zps_eAplZdoTransportNwkKey()**, which requires the associated key sequence number to be specified
- Requested from the Trust Centre by calling the function **zps_eAplZdoRequestKeyReq()** on the node that needs the network key

On reaching its destination(s), the transported key is automatically saved but not activated. A stored network key can be adopted as the active key using the function **zps_eAplZdoSwitchKeyReq()**, which is called on the Trust Centre and which identifies the required key by means of its unique sequence number.

5.8.4.2 Application Link Key Modification

An application link key can be introduced or replaced by the application using **zps_eAplZdoAddReplaceLinkKey()**. If a link key already exists for the same node-pair, it will be replaced by the new link key. The function must be called on both nodes in the pair.

5.9 Using Support Software Features

This section describes certain support software features and how to include them in your application code:

- Message queues are described in Section 5.9.1
- Software timers are described in Section 5.9.2

The referenced API resources are detailed in Chapter 9.

5.9.1 Message Queues

Communications between application tasks on a node are implemented via message queues. The application can create a dedicated message queue for a particular communication channel. A set of functions are provided to implement message queues, as indicated in Section 5.9.1.1 below (these functions are detailed in Section 9.1). The stack requires certain standard queues, as indicated in Section 5.9.1.2 below.

> ***Caution:*** *To allow the device to enter sleep mode, the message queues must not contain any messages. All message queues must first be emptied.*

5.9.1.1 General Queue Management

A queue can be created using the function **ZQ_vZQueueCreate()**. This function allows the queue size (number of messages that it can hold) and the size of a message to be specified. A queue is given a unique handle, which is a pointer to a `tszQueue` structure containing up-to-date information about the queue (see Section 9.1.2.1).

A message can be placed in a (created) queue using the function **ZQ_bZQueueSend()** and a message can be retrieved from a queue using the function **ZQ_bZQueueReceive()**. This is illustrated in Figure 13 below.

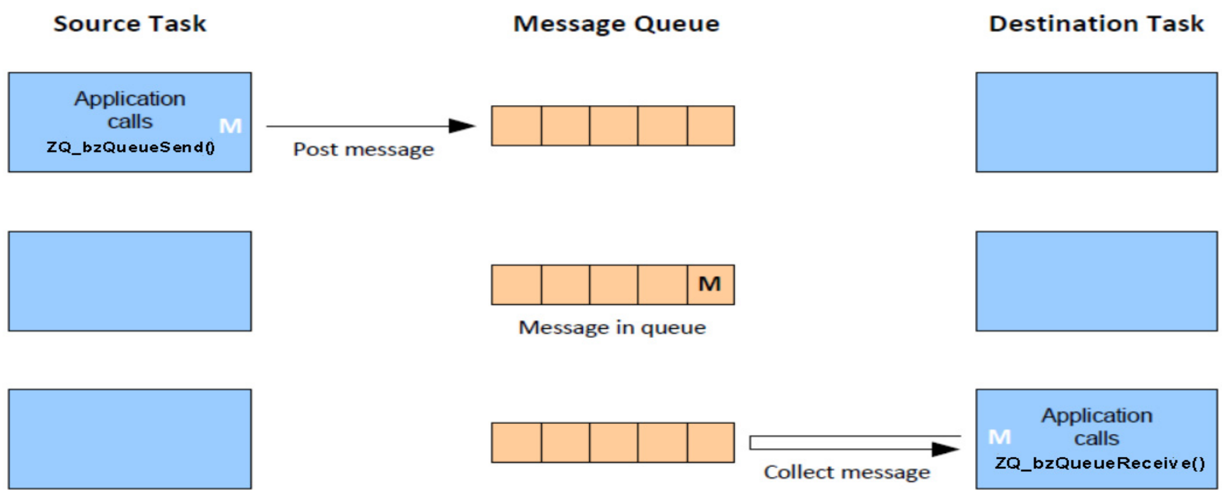


Figure 13: Sending/Receiving a Message via a Message Queue

When the above two functions are called, the `tszQueue` structure for the queue is automatically updated to reflect the new state of the queue. Retrieving a message results in the message being deleted from the queue.

The application must regularly poll a message queue through which it expects to receive messages. It can do this by periodically calling the **ZQ_bQueueIsEmpty()** function, which checks whether the queue is empty. If the queue is not empty, it should call **ZQ_bZQueueReceive()** until there are no more messages in the queue. The number of messages currently waiting to be collected from the queue can be obtained using the function **ZQ_u32QueueGetQueueMessageWaiting()**.

5.9.1.2 Standard Stack Queues

Three standard queues must be created by the application for use by the stack:

- Queue with handle zps_msgMlmeDcfmInd to receive IEEE 802.15.4 MAC command packets from other nodes
- Queue with handle zps_msgMcpsDcfmInd to receive IEEE 802.15.4 MAC data packets from other nodes
- Queue with handle zps_TimeEvents to receive internal software timer events (such as a timer expiry event)

Example code for the creation of these queues is provided below:

```
ZQ_vZQueueCreate(&zps_msgMlmeDcfmInd, MLME_QUEUE_SIZE,
              sizeof(MAC_tsMlmeVsDcfmInd),(uint8*)asMacMlmeVsDcfmInd);
ZQ_vZQueueCreate(&zps_msgMcpsDcfmInd, MCPS_QUEUE_SIZE,
              sizeof(MAC_tsMcpsVsDcfmInd),(uint8*)asMacMcpsDcfmInd);
ZQ_vQueueCreate(&zps_TimeEvents,TIMER_QUEUE_SIZE,sizeof(zps_tsTimeEvent),
              (uint8*)asTimeEvent);
```

You simply need to include the above code in your application. You do not need to process these queues in your code.

More information on the receive queues is provided in Appendix B.6.

5.9.2 Software Timers

The ZigBee 3.0 SDK provides resources that allow an application to implement and interact with software timers on the local node. Multiple software timers can be used concurrently and they are all derived from the same source counter, which is the ZigBee Tick Timer.

> ***Caution:*** *To allow the device to enter sleep mode, no software timers should be active. Any running software timers must first be stopped and all timers must be closed.*

5.9.2.1 Setting Up Timers

To set up software timers in your application code, you must:

- Declare an array of `ZTIMER_tsTimer` structures (see Section 9.2.2.1), where each element/structure contains information on one timer
- Call the function **ZTIMER_vTask()** in the while loop of your application - this allows the stack software to automatically update the structure for each timer as the timer runs

For each timer, a user-defined callback function must be provided, which is referenced from the timer's structure. This callback function, **ZTIMER_tpfCallback()**, is called when the timer expires (reaches its timed period) in order to perform any operations that the application requires as a result of the timer expiration.

Before any of the software timers can be used, they must be collectively initialised by calling the function **ZTIMER_eInit()**. This function takes the array of timer structures as an input.

Before an individual timer can be used, it must be opened using the function **ZTIMER_eOpen()**. Similarly, when the timer is no longer required, it should be closed using the function **ZTIMER_eClose()**. A timer is specified in these functions by means of its index in the array of timer structures.

5.9.2.2 Operating Timers

Once an individual software timer has been opened, it can be run one or more times before it is closed. A timer can be run by calling the function **ZTIMER_eStart()**. The timed period must be specified in milliseconds. On expiration of the timer, the user-defined callback function **ZTIMER_tpfCallback()** will be called to perform any operations required by the application.

A running timer can be stopped before it expires by calling the function **ZTIMER_eStop()**.

The status of an individual timer can be obtained at any time using the function **ZTIMER_eGetState()**. The possible reported states are Running, Stopped, Expired and Closed.

5.9.3 Critical Sections and Mutual Exclusion (Mutex)

The ZigBee 3.0 stack software provides features to prevent sections of application code from being pre-empted and/or re-entered. For example, when the application is writing data to memory, it may not be desirable for this operation to be interrupted and for an interrupt service routine to start writing to the same memory block.

Two features are provided to protect sections of application code:

- **Critical Section:** A section of application code can be designated as a ‘critical section’, which means that the execution of this code section cannot be pre-empted by an interrupt with a priority level less than 12. A critical section should be short in order to avoid suspending interrupts for a long period of time.
- **Mutual Exclusion (Mutex):** It may be desirable for a section of code not to be re-entrant. A ‘mutex’ can be associated with a code section to prevent it from being entered again before the current execution of the section has completed.

These features are described in more detail in the sub-sections below. The API resources to implement these features are detailed in Section 9.3.

5.9.3.1 Implementing a Critical Section

The execution of a critical section of application code cannot be pre-empted by an interrupt with a priority level less than 12 (higher-priority interrupts can always pre-empt a critical section). This is illustrated in Figure 14 below, which shows the interplay between the main application thread and an interrupt service routine (ISR).

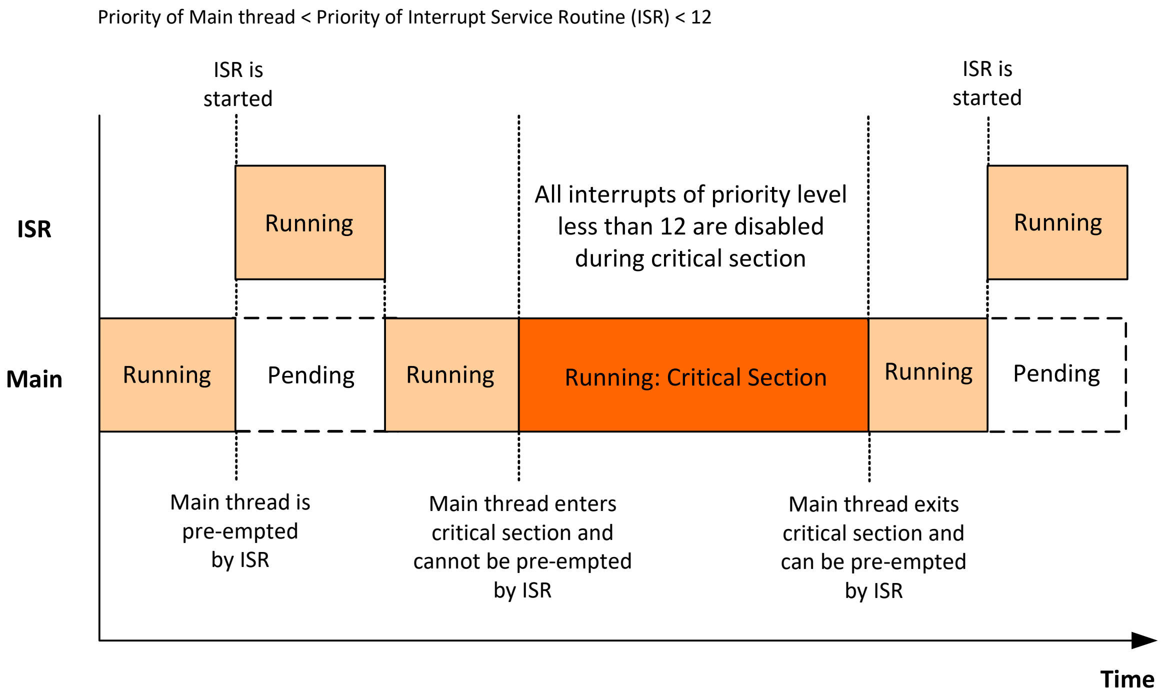


Figure 14: Critical Section Illustration

A critical section of code must be delimited by the following two functions:

- **zps_eEnterCriticalSection()** must be called at the start of the critical section.
- **zps_eExitCriticalSection()** must be called at the end of the critical section.

A mutex can also be optionally associated with a critical section, to protect the section from re-entrancy. If required, the mutex can be specified in a parameter of **zps_eEnterCriticalSection()**. Mutexes are described in Section 5.9.3.2.

To implement critical sections, the application must maintain a 'priority level' value `u8Level` (see Section 9.3.2.1) which contains the current priority level of the main application thread (when critical sections are not being executed). When a critical section is entered, the priority level of the main thread is increased such that interrupts with a priority of 11 or less cannot pre-empt the main thread. At the end of the critical section, the priority level of the main thread is returned to the value that was contained in `u8Level` before the critical section was entered.

5.9.3.2 Implementing a Mutex

A mutex can be associated with a section of application code to prevent the section from being re-entered before the current execution of the section has finished. The section of code to which the mutex will be applied must be delimited by the following two functions:

- **zps_u8GrabMutexLock()** must be called at the start of the code section.
- **zps_u8ReleaseMutexLock()** must be called at the end of the code section.

It is also possible to apply a mutex to a critical section, as described in Section 5.9.3.1.

When applying a mutex, a pointer must be provided to a user-defined mutex function with the following prototype:

((bool_t*) (*) (void))

This function must define and maintain a Boolean flag which indicates whether the corresponding mutex is active (TRUE) or inactive (FALSE). This flag is used by the API functions to determine whether the specified mutex is available. If this flag reads as FALSE at the start of the relevant code section, the mutex is applied and the above mutex function must set the flag to TRUE, but if the flag is already TRUE then the mutex cannot be applied (and the API function returns with a failure).

To implement mutex protection, the application must maintain a 'priority level' value `u8Level` (see Section 9.3.2.1) which contains the current priority level of the main application thread (when mutex-protected sections are not being executed). When a mutex is applied, the priority level of the main thread is increased such that interrupts with a priority of 11 or less cannot pre-empt the main thread. When the mutex is released, the priority level of the main thread is returned to the value that was contained in `u8Level` before the mutex was applied.

5.10 Advanced Features

This section describes the implementation of advanced ZigBee features that have been introduced in ZigBee 3.0.

5.10.1 End Device Ageing

A Router which is a parent needs to maintain its Neighbour table, which involves discarding inactive children (that may have left the network) in order to make way for potential new children. An End Device Ageing mechanism is available to support this maintenance.

In this mechanism, a timeout is applied to every child entry in the Router's Neighbour table. If a packet, called a 'keep-alive' packet, is not received from an End Device child before its timeout expires, the child is assumed to be no longer active and is removed from the table (and therefore from the Router's children).

Timeout Period

The timeout period is specific to an individual child and is set on the End Device using the function **zps_bAplAfSetEndDeviceTimeout()**. This period is communicated to the parent via an End Device Timeout Request when the End Device joins (or re-joins) the network. The timeout is applied by the Router to the Neighbour table entry for the End Device. The arrival of a keep-alive packet from the End Device will result in the timeout being re-started from the beginning. If the timeout is allowed to expire (without a keep-alive packet), the Router will delete the relevant child entry from the Neighbour table.

> **Note 1:** The Router initially sets the timeout for all End Device children to the default value defined in the NIB, which is 256 minutes in the NXP software. The timeout will remain at this value unless changed by the End Device, as described above.
>
> **Note 2:** After receiving the End Device Timeout Request, the parent will send an End Device Timeout Response to the End Device, indicating the outcome of the request. If the request has been successful, the End Device can subsequently send keep-alive packets.

Keep-alive Packets

A keep-alive packet can be sent from the End Device using the function **zps_eAplAfSendKeepAlive()**. It is recommended that this function is called at least three times within the timeout period defined for the End Device, in order to prevent the child from being accidentally removed from the network due to missed keep-alive packets at the parent.

A keep-alive packet can be either of the following types:

- **MAC Data Poll:** In this case, the parent may send pending data back to the End Device. The arrival of this data at the End Device will be indicated by a zps_EVENT_AF_DATA_INDICATION event (as described in Section 5.5.2).
- **End Device Timeout Request**: This packet type simply has the effect of re-starting the timeout for the End Device on the parent, which will return an End Device Timeout Response to the End Device, indicating the outcome of the request.

The keep-alive packet type to be used is determined by the Router parent and is configured in the NIB on the parent - in the NXP software, a Router is configured to accept either packet type, by default. This information is communicated to the End Device in the initial End Device Timeout Response that is sent to the End Device on joining the network. The **zps_eAplAfSendKeepAlive()** function will then automatically send the appropriate keep-alive packet type - where either packet type is accepted by the parent, the function sends a Data Poll packet.

5.10.2 Distributed Security Networks

In a traditional ZigBee network, security is implemented by a Trust Centre, which is normally the Co-ordinator - this uses a centralised security scheme. In a distributed security network, any Router node can manage security and so security management is distributed throughout the network. A distributed security network does not have a Co-ordinator/Trust Centre, and consists only of Routers and End Devices - any Router can create the network.

In a distributed security network, only network-level security can be implemented. A network key is generated by the Router that creates the network (as described in Section 5.8.3) and is passed on to other nodes, including other Routers, as the network grows. During this distribution, the network key is encrypted using a ‘Distributed Security Global Link Key’, which is a type of pre-configured global link key (see Section 5.8.2).

A distributed security network can be started on a Router node using the function **zps_eAplFormDistributedNetworkRouter()**. The start parameters are specified through a `zps_tsAftsStartParamsDistributed` structure (see Section 7.2.3.7). These parameters include:

- PAN ID
- Extended PAN ID (EPID)
- Radio channel
- Pointer to a location to receive the network key

This first node of the network will generate the network key (saved to the above location) and pass this key to nodes that join it.

The function **zps_eAplFormDistributedNetworkRouter()** can also be called on other Router nodes to join them to the network. An End Device can be joined to a distributed network using the function **zps_eAplInitEndDeviceDistributed()**. However, these nodes are more likely to be introduced to the network via other commissioning methods, such as Touchlink and NFC commissioning.

5.10.3 Filtering Packets on LQI Value/Link Cost

This section describes the operation and configuration of the filtering of received data packets based on LQI value (detected signal strength). Packet filtering results in some received packets with low LQI values being discarded.

In practice, the measured LQI values of packets are translated into 'link cost' values for filtering, as detailed in Section 5.10.3.1.

Packet filtering is optional and can be beneficial during:

- network joining
- route discovery
- normal network operation

The operation and benefits of packet filtering are described in Section 5.10.3.2.

Packet filtering can be enabled using the function **zps_vAplAfEnableMcpsFilter()** and modified as described in Section 5.10.3.3.

5.10.3.1 Link Cost

For the purpose of packet filtering, LQI values are translated into 'link cost' values. Thus, a range of LQI values maps to a single link cost, which is an integer value. The default mappings implemented by the ZigBee PRO stack are shown in Table 5 below.

LQI Range	Link Cost
≥ 51	1
46 — 50	2
41 — 45	3
39 — 40	4
36 — 38	5
25 — 35	6
≤ 24	7

Table 5: 'LQI to Link Cost' Mappings

The above mappings can be modified, as described in Section 5.10.3.3.

A link cost of 5 is used as the packet filtering threshold by the NXP ZigBee PRO stack. Thus, packets with link costs greater than 5 may be discarded. For the device, this threshold is more suitable than the value of 3 proposed in the ZigBee specification. However, the threshold is configurable, as described in Section 5.10.3.3.

5.10.3.2 Packet Filtering in Operation

Packet filtering is an optional feature of the ZigBee PRO stack that is applied by the IEEE 802.15.4 MAC layer. It is useful during network joining, route discovery and normal network operation to optimise the processing of received packets.

Network Joining

During network joining, a form of packet filtering is applied to the results of the network discovery phase. Any potential parents that have been discovered are filtered such that nodes with link costs greater than 5 (low LQI values) are discarded. This feature aids the formation of networks with strong links between neighbours and is most effective in dense networks. For more information about this process during network joining, refer to the ZigBee specification.

Route Discovery and Normal Network Operation

In a large network, traffic levels are high during both route discovery and normal operation, and a node is likely to receive many data packets. There is, however, limited storage capacity on a node to hold these packets until they can be processed. To restrict the number of received packets that are submitted to the receive queue, the following filtering system is applied:

- All unicast packets are queued (without filtering) provided that sufficient space is available in the receive queue.
- Broadcast packets are queued provided that at least 50% of the receive queue capacity is free, otherwise the packet filtering mechanism is applied and only packets with a link cost of 5 or less are queued.

During route discovery, this filtering prevents nodes with low associated LQI values from being entered into the Neighbour table, allowing reliable routes to be established. For example, it may be more desirable to establish a route comprising multiple hops with good LQI values than a single hop with a poor LQI value.

5.10.3.3 Packet Filtering Configuration

Packet filtering is disabled by default but can be enabled and re-configured as described below.

Basic Configuration

The function **zps_vAplAfEnableMcpsFilter()** allows the stack's packet filtering to be enabled and the link cost threshold to be adjusted (from the default value of 5). This function is detailed in Section 7.1.1. If required, it can be called at any time after **zps_eAplAfInit()**.

Link Cost Configuration

The mappings between LQI values and link costs can be modified from the default mappings detailed in Section 5.10.3.1. To modify the mappings, the following function must be user-defined, which translates an LQI value (input) into a link cost (output):

uint8 APP_u8LinkCost(uint8 *u8Lqi***);**

An example function which implements the default mappings is shown below:

```
PRIVATE uint8 APP_u8LinkCost ( uint8 u8Lqi )
{
    uint8 u8Lc;
    if (u8Lqi > 50)
        {
            u8Lc = 1;
        }
        else if ((u8Lqi <= 50) && (u8Lqi > 45))
        {
            u8Lc = 2;
        }
        else if ((u8Lqi <= 45) && (u8Lqi > 40))
        {
            u8Lc = 3;
        }
        else if ((u8Lqi <= 40) && (u8Lqi > 38))
        {
            u8Lc = 4;
        }
        else if ((u8Lqi <= 38) && (u8Lqi > 35))
        {
            u8Lc = 5;
        }
        else if ((u8Lqi <= 35) && (u8Lqi > 24))
        {
           u8Lc = 6;
        }
        else
        {
           u8Lc = 7;
        }
    return u8Lc;
}
```

The above function must be registered as a callback function using the following callback registration function **zps_vNwkLinkCostCallbackRegister()**, which is detailed in Section 7.1.1. This function takes a pointer to the **APP_u8LinkCost()** function to be registered. If required, the registration function must be called before **zps_eAplAfInit()**, and on both cold and warm starts.

5.10.4 Device Permissions

The function **zps_eAplZdoSetDevicePermission()** allows certain permissions to be set on the local device. These permissions are as follows:

Enumeration	Description
zps_DEVICE_PERMISSIONS_ALL_PERMITED	Allow all requests from other nodes
zps_DEVICE_PERMISSIONS_JOIN_DISALLOWED	Do not allow join requests from other nodes
zps_DEVICE_PERMISSIONS_DATA_REQUEST_DISALLOWED	Do not allow data requests from other nodes and disable end-to-end acknowledgements

Table 6: Device Permissions

When a device joins the network, the ALL_PERMITED option is set by default, so the device can respond to requests from other nodes.

However, if the network employs security set up using the ZigBee Key Establishment cluster (e.g. a Smart Energy network), it is necessary to disallow data requests and end-to-end acknowledgements on the newly joined node during the key establishment process. The application must do this as follows:

1. Once an event has occurred to indicate that the device has joined the network (the event zps_EVENT_NWK_JOINED_AS_ROUTER or zps_EVENT_NWK_JOINED_AS_ENDDEVICE), the application must disallow data requests and APS end-to-end acknowledgements by calling **zps_eAplZdoSetDevicePermission()** with the option DATA_REQUEST_DISALLOWED.
2. The key establishment process can then be started using the function provided for the Key Establishment cluster.
3. Once the key establishment process has successfully completed, data requests and APS end-to-end acknowledgements can be allowed again by calling **zps_eAplZdoSetDevicePermission()** with the ALL_PERMITED option.

The key establishment process and associated resources are fully described in the documentation for the Key Establishment cluster (e.g. in the *ZigBee Smart Energy User Guide)*.

Part II: Reference Information

6. ZigBee Device Objects (ZDO) API

The chapter describes the resources of the ZigBee Device Objects (ZDO) API. This API is primarily concerned with starting, forming and modifying a ZigBee PRO network. The API is defined in the header file **zps_apl_zdo.h**.

In this chapter:

- Section 6.1 details the ZDO API functions
- Section 6.2 details the ZDO API enumerations

6.1 ZDO API Functions

The ZDO API functions are divided into the following categories:

- **Network Deployment** functions, described in Section 6.1.1
- **Security** functions, described in Section 6.1.2
- **Addressing** functions, described in Section 6.1.3
- **Routing** functions, described in Section 6.1.4
- **Object Handle** functions, described in Section 6.1.5
- **Optional Cluster** function, described in Section 6.1.6

6.1.1 Network Deployment Functions

The ZDO Network Deployment functions are used to start the ZigBee PRO stack, and allow devices to join the network and bind to each other, as well as leave the network.

The functions are listed below, along with their page references:

Note: The ZDO initialisation and start stack functions use network parameter values that have been pre-set and saved using the ZPS Configuration Editor - see Chapter 12.

ZPS_eAplZdoStartStack

```
ZPS_teStatus ZPS_eAplZdoStartStack(void);
```

Description

This function starts the ZigBee PRO stack. The steps taken depend on the node type:

- If the device is the Co-ordinator, this function will start the network formation process.
- If the device is a Router or End Device, this function will start the network discovery process - that is, the device will search for a network to join.

When the stack starts, the 2400-MHz radio channel to be used by the device is selected. The channels (in the range 11 to 26) available to the device should be specified in advance using the ZPS Configuration Editor (see Chapter 12) and can be either of the following:

- A fixed channel
- A set of channels for a channel scan:
  - If the device is the Co-ordinator, this is the set of channels that the device will scan to find a suitable operating channel for the network.
  - If the device is a Router or End Device, this is the set of channels that the device will scan to find a network to join.

If this function successfully initiates network formation or discovery, ZPS_E_SUCCESS will be returned. Subsequent results from this process will then be reported through stack events (see Section 10.1 for details of these events):

- If the Co-ordinator successfully creates a network, the event ZPS_EVENT_NWK_STARTED is generated. Otherwise, the event ZPS_EVENT_NWK_FAILED_TO_START is generated.
- When the network discovery process for a Router or End Device has completed, the subsequent actions depend on the Extended PAN ID (EPID) that has been pre-set using the ZPS Configuration Editor:
  - If a zero EPID value was pre-set, the stack event ZPS_EVENT_NWK_DISCOVERY_COMPLETE is generated. This includes a list of the detected networks and the index (in the list) of the recommended network to join. You can then call **ZPS_eAplZdoJoinNetwork()** to join the desired network.
  - If a non-zero EPID value was pre-set, the device will automatically attempt to join the network with this EPID, provided that such a network has been discovered. Note that the 'permit joining' setting of the potential parent will be ignored.

The maximum depth (number of levels below the Co-ordinator) of the network is 15.

Parameters

None

Returns

ZPS_E_SUCCESS (stack started and network formation/discovery begun)

APS return codes, listed and described in Section 10.2.2

NWK return codes, listed and described in Section 10.2.3

MAC return codes, listed and described in Section 10.2.4

ZPS_vDefaultStack

```
void ZPS_vDefaultStack(void);
```

Description

This function can be used to reset the ZigBee PRO stack to its default state. It removes previous context data for the stack, but leaves NWK layer frame counters intact.

Note: After calling this function, all security keys must be re-configured.

Parameters

None

Returns

None

ZPS_eAplZdoGetDeviceType

```
ZPS_teZdoDeviceType ZPS_eAplZdoGetDeviceType(void);
```

Description

This function can be used to obtain the ZigBee node type (Co-ordinator, Router or End Device) of the local node.

Parameters

None

Returns

ZigBee node type, one of:

ZPS_ZDO_DEVICE_COORD (Co-ordinator)
ZPS_ZDO_DEVICE_ROUTER (Router)
ZPS_ZDO_DEVICE_ENDDEVICE (End Device)

ZPS_eAplZdoDiscoverNetworks

```
ZPS_teStatus ZPS_eAplZdoDiscoverNetworks(
                         uint32 u32ChannelMask);
```

Description

This function can be used by a Router or End Device to initiate a network discovery - that is, to find a network to join.

A network discovery is performed when the stack is started using the function **ZPS_eAplZdoStartStack()**. The function **ZPS_eAplZdoDiscoverNetworks()** can be used to perform subsequent network discoveries (for example, if the initial search did not yield any suitable networks).

As part of this function call, you must specify a value which indicates the 2400-MHz radio channels (numbered 11 to 26) to be used in the network search. There are two ways of setting this parameter:

- A single value in the range 11 to 26 can be specified, indicating that the corresponding channel (and no other) must be used - for example, 12 indicates use channel 12.
- A 32-bit mask can be used to specify a set of channels that the device will scan to find a network - each of bits 11 to 26 represents the corresponding radio channel, where the channel will be included in the scan if the bit is set to 1 (and excluded if cleared to 0). Therefore, the value 0x07FFF800 represents all channels.

Note that if an invalid value is specified for this parameter, the default value of 0x07FFF800 (all channels) will be used.

If this function successfully initiates a network discovery, ZPS_E_SUCCESS will be returned. The network discovery results will then be reported through the event ZPS_EVENT_NWK_DISCOVERY_COMPLETE (for details of this event, refer to Section 7.2.2.9). This includes a list of the detected networks and the index (in the list) of the recommended network to join. You should then call **ZPS_eAplZdoJoinNetwork()** to join the desired network.

Parameters

u32ChannelMask	Radio channel(s) for network discovery (see above)

Returns

ZPS_E_SUCCESS (network discovery started)

APS return codes, listed and described in Section 10.2.2

NWK return codes, listed and described in Section 10.2.3

MAC return codes, listed and described in Section 10.2.4

ZPS_eAplZdoJoinNetwork

```
ZPS_teStatus ZPS_eAplZdoJoinNetwork(
                    ZPS_tsNwkNetworkDescr *psNetworkDescr);
```

Description

This function can be used by a Router or End Device to send a request to join a particular network, following a network discovery.

Note: This function is not needed if the network to join has been pre-determined by setting the advanced device parameter *APS Use Extended PAN Id* using the ZPS Configuration Editor. In this case, a join will be attempted automatically after starting the stack.

The required network is specified using its network descriptor, obtained in a ZPS_EVENT_NWK_DISCOVERY_COMPLETE event which results from a network discovery previously implemented using **ZPS_eAplZdoStartStack()** or **ZPS_eAplZdoDiscoverNetworks()**. For details of this event, refer to Section 7.2.2.9.

If the join request is successfully sent, the function will return ZPS_E_SUCCESS (note that this does not mean that device has joined the network). The result of the join request will then be reported through a stack event (see Section 10.1 for details of these events):

- If the device successfully joined the network as a Router, the event ZPS_EVENT_NWK_JOINED_AS_ROUTER is generated. The allocated 16-bit network address of the Router is returned as part of this stack event.
- If the device successfully joined the network as an End Device, the event ZPS_EVENT_NWK_JOINED_AS_ENDDEVICE is generated. The allocated 16-bit network address of the End Device is returned as part of this stack event.
- If the join request was unsuccessful, the event ZPS_EVENT_NWK_FAILED_TO_JOIN is generated.

Note that nodes can join a ZigBee PRO network to a maximum depth of 15 (levels below the Co-ordinator).

Parameters

**psNetworkDescr*	Pointer to network descriptor of network to join

Returns

ZPS_E_SUCCESS (join request successfully sent)

APS return codes, listed and described in Section 10.2.2

NWK return codes, listed and described in Section 10.2.3

MAC return codes, listed and described in Section 10.2.4

ZPS_eAplZdoRejoinNetwork

```
ZPS_teStatus ZPS_eAplZdoRejoinNetwork(
                                    bool_t bWithDiscovery);
```

Description

This function can be used by an active Router or End Device to send a request to rejoin its previous network. The function should be called if the application detects that it has lost its connection to the network - this is indicated by an excessive number of failed communications (for example, with many missing acknowledgements).

Options are provided to first perform a network discovery to find potential parents to join or simply rejoin the previous parent.

If the rejoin request is successfully sent, the function will return ZPS_E_SUCCESS (note that this does not mean that device has rejoined the network). The result of the rejoin request will then be reported through a stack event (see Section 10.1 for details of these events):

- If the device successfully rejoined the network as a Router, the event ZPS_EVENT_NWK_JOINED_AS_ROUTER is generated.
- If the device successfully rejoined the network as an End Device, the event ZPS_EVENT_NWK_JOINED_AS_ENDDEVICE is generated.
- If the rejoin request was unsuccessful, the event ZPS_EVENT_NWK_FAILED_TO_JOIN is generated.

In the case of a successful rejoin, the node will retain its previously allocated 16-bit network address.

Note that the 'permit joining' status of the potential parent is ignored during a rejoin.

Parameters

bWithDiscovery	Specifies whether a network discovery is required: TRUE - perform network discovery before rejoining FALSE - rejoin previous parent

Returns

ZPS_E_SUCCESS (rejoin request successfully sent)

APS return codes, listed and described in Section 10.2.2

NWK return codes, listed and described in Section 10.2.3

MAC return codes, listed and described in Section 10.2.4

ZPS_eAplZdoDirectJoinNetwork

```
ZPS_teStatus ZPS_eAplZdoDirectJoinNetwork(
                                    uint64 u64Addr,
                                    uint16 u16Addr,
                                    uint8 u8Capability);
```

Description

This function can be used on a Router and on the Co-ordinator to pre-determine the child nodes that will directly join it. The function is called to register each child node separately, and the IEEE/MAC and network addresses of the child node must be specified.

The function adds the registered node to its Neighour table (it actually adds the node's IEEE/MAC address to the MAC Address table and then includes the index of this address in a Neighbour table entry for the node). The function must be called only when the parent node is fully up and running in the network. Since the child node has not yet joined the network but is in the Neighbour table, it will be perceived by the parent as having been orphaned. Therefore, when the child node attempts to join the network, it must perform a rejoin as an orphan by calling the function **ZPS_eAplZdoOrphanRejoinNetwork()**.

> ***Caution:*** *You should only modify to the Neighbour table using this function and never write to it directly.*

Parameters

u64Addr	IEEE/MAC address of child node to be registered
u16Addr	Network address of child node to be registered
u8Capability	A bitmap indicating the operational capabilities of the child node - this bitmap is detailed in Table 8 on page 199

Returns

ZPS_E_SUCCESS
(child node successfully registered)
ZPS_APL_APS_E_ILLEGAL_REQUEST
(address 0x0, address 0xFFFFFFFFFFFFFFFF, own address, ZDO busy)
ZPS_NWK_ENUM_ALREADY_PRESENT
ZPS_NWK_ENUM_NEIGHBOR_TABLE_FULL

ZPS_eAplZdoOrphanRejoinNetwork

```
ZPS_teStatus ZPS_eAplZdoOrphanRejoinNetwork(void);
```

Description

This function can be used by an orphaned node to attempt to rejoin the network - the orphaned node may be an End Device or a Router. The function should also be used for a first-time join for which the parent has been pre-determined using the function **ZPS_eAplZdoDirectJoinNetwork()**.

The function starts the stack on the node. Therefore, when this function is used, there is no need to explicitly start the stack using **ZPS_eAplZdoStartStack()**.

If the rejoin request is successfully sent, the function will return ZPS_E_SUCCESS (note that this does not mean that device has rejoined the network). The result of the rejoin request will then be reported through a stack event (see Section 10.1 for details of these events):

- If the device successfully rejoined the network as a Router, the event ZPS_EVENT_NWK_JOINED_AS_ROUTER is generated.
- If the device successfully rejoined the network as an End Device, the event ZPS_EVENT_NWK_JOINED_AS_ENDDEVICE is generated.
- If the rejoin request was unsuccessful, the event ZPS_EVENT_NWK_FAILED_TO_JOIN is generated.

In the case of a successful rejoin of a genuinely orphaned node, the node will retain its previously allocated 16-bit network address.

Note that the ‘permit joining’ status of the potential parent is ignored during a rejoin.

Parameters

None

Returns

ZPS_E_SUCCESS
(rejoin request successfully sent)
ZPS_APL_APS_E_ILLEGAL_REQUEST
(missing EPID, called from Co-ordinator, ZDO busy)

ZPS_eAplZdoPermitJoining

```
ZPS_teStatus ZPS_eAplZdoPermitJoining(
                                    uint8 u8PermitDuration);
```

Description

This function can be used on a Router or the Co-ordinator to control whether new child nodes are allowed to join it - that is, to set the node's 'permit joining' status. The function can be used to enable joining permanently or for a fixed duration, or to disable joining (permanently).

The specified parameter value determines the 'permit joining' status, as follows:

0	Disables joining
1 - 254	Enables joining for specified time interval, in seconds
255	Enables joining permanently

For example, if the parameter is set to 60, joining will be enabled for the next 60 seconds and then automatically disabled.

Caution: *The 'permit joining' setting of a device is ignored during a join attempt in which a non-zero Extended PAN ID is specified on the joining device and during any rejoin attempt.*

Parameters

u8PermitDuration	Time duration, in seconds, for which joining will be permitted (see above)

Returns

ZPS_E_SUCCESS ('permit joining' status successfully set)

APS return codes, listed and described in Section 10.2.2

NWK return codes, listed and described in Section 10.2.3

MAC return codes, listed and described in Section 10.2.4

ZPS_u16AplZdoGetNetworkPanId

```
uint16 ZPS_u16AplZdoGetNetworkPanId(void);
```

Description

This function obtains the 16-bit PAN ID of the ZigBee network to which the local node currently belongs.

Parameters

None

Returns

PAN ID of current network

ZPS_u64AplZdoGetNetworkExtendedPanId

uint64 ZPS_u64AplZdoGetNetworkExtendedPanId(void)

Description

This function obtains the 64-bit Extended PAN ID (EPID) of the ZigBee PRO network to which the local node currently belongs.

Parameters

None

Returns

Extended PAN ID of current network

ZPS_u8AplZdoGetRadioChannel

```
uint8 ZPS_u8AplZdoGetRadioChannel(void);
```

Description

This function obtains the 2400-MHz band channel in which the local node is currently operating. The channel is represented by an integer in the range 11 to 26.

Parameters

None

Returns

Radio channel number (in range 11-26)

ZPS_eAplZdoBind

```
ZPS_teStatus ZPS_eAplZdoBind(
                    uint16 u16ClusterId,
                    uint8 u8SrcEndpoint,
                    uint16 u16DstAddr,
                    uint64 u64DstIeeeAddr,
                    uint8 u8DstEndpoint);
```

Description

This function requests a binding to be created between an endpoint on the local node and an endpoint on a remote node. The source endpoint and cluster must be specified, as well as the destination node and endpoint. The destination node is specified using both its 64-bit IEEE (MAC) address and its 16-bit network address.

The binding is added to the binding table on the local node.

A binding to multiple remote endpoints (collected into a group) can be created using the function **ZPS_eAplZdoBindGroup()**.

Parameters

u16ClusterId	Identifier of cluster on source node to be bound
u8SrcEndpoint	Number of endpoint (1-240) on source node to be bound
u16DstAddr	16-bit network address of destination for binding
u64DstIeeeAddr	64-bit IEEE (MAC) address of destination for binding
u8DstEndpoint	Number of endpoint on destination node to be bound

Returns

ZPS_E_SUCCESS (binding successfully created)

APS return codes, listed and described in Section 10.2.2

NWK return codes, listed and described in Section 10.2.3

MAC return codes, listed and described in Section 10.2.4

ZPS_eAplZdoUnbind

ZPS_teStatus ZPS_eAplZdoUnbind(
uint16 *u16ClusterId*,
uint8 *u8SrcEndpoint*,
uint16 *u16DstAddr*,
uint64 *u64DstIeeeAddr*,
uint8 *u8DstEndpoint***);**

Description

This function requests an existing binding to be removed between an endpoint on the local node and an endpoint on a remote node, where this binding was created using the function **ZPS_eAplZdoBind()**. The source endpoint and cluster must be specified, as well as the destination node and endpoint. The destination node is specified using both its 64-bit IEEE (MAC) address and its 16-bit network address.

The binding is removed from the binding table on the local node.

Parameters

u16ClusterId	Identifier of bound cluster on source node
u8SrcEndpoint	Number of bound endpoint (1-240) on source node
u16DstAddr	16-bit network address of destination for binding
u64DstIeeeAddr	64-bit IEEE (MAC) address of destination for binding
u8DstEndpoint	Number of bound endpoint on destination node

Returns

ZPS_E_SUCCESS (binding successfully removed)

APS return codes, listed and described in Section 10.2.2

NWK return codes, listed and described in Section 10.2.3

MAC return codes, listed and described in Section 10.2.4

ZPS_eAplZdoBindGroup

```
ZPS_teStatus ZPS_eAplZdoBindGroup(
                                  uint16 u16ClusterId,
                                  uint8 u8SrcEndpoint,
                                  uint16 u16DstGrpAddr);
```

Description

This function requests a binding to be created between an endpoint on the local node and multiple endpoints on remote nodes. The source endpoint and cluster must be specified, as well as the destination nodes/endpoints for the binding, which must be specified using a 16-bit group address, previously set up using **ZPS_eAplZdoGroupEndpointAdd()**.

The binding is added to the binding table on the local node.

Parameters

u16ClusterId	Identifier of cluster on source node to be bound
u8SrcEndpoint	Number of endpoint (1-240) on source node to be bound
u16DstGrpAddr	16-bit group address of destination group for binding

Returns

ZPS_E_SUCCESS (binding successfully created)

APS return codes, listed and described in Section 10.2.2

NWK return codes, listed and described in Section 10.2.3

MAC return codes, listed and described in Section 10.2.4

ZPS_eAplZdoUnbindGroup

```
ZPS_teStatus ZPS_eAplZdoUnbindGroup(
                                    uint16 u16ClusterId,
                                    uint8 u8SrcEndpoint,
                                    uint16 u16DstGrpAddr);
```

Description

This function requests an existing binding to be removed between an endpoint on the local node and a group of endpoints on remote nodes, where this binding was created using the function **ZPS_eAplZdoBindGroup()**. The source endpoint and cluster must be specified, as well as the destination nodes/endpoints for the binding, which must be specified using a 16-bit group address.

The binding is removed from the binding table on the local node.

Parameters

u16ClusterId	Identifier of bound cluster on source node
u8SrcEndpoint	Number of bound endpoint (1-240) on source node
u16DstGrpAddr	16-bit group address of bound destination group

Returns

ZPS_E_SUCCESS (binding successfully removed)

APS return codes, listed and described in Section 10.2.2

NWK return codes, listed and described in Section 10.2.3

MAC return codes, listed and described in Section 10.2.4

ZPS_ePurgeBindTable

ZPS_teStatus ZPS_ePurgeBindTable(void);

Description

This function removes all bindings from the binding table on the local node.

Parameters

None

Returns

ZPS_E_SUCCESS (binding successfully removed)
APS return codes, listed and described in Section 10.2.2
NWK return codes, listed and described in Section 10.2.3
MAC return codes, listed and described in Section 10.2.4

ZPS_eAplZdoPoll

```
ZPS_teStatus ZPS_eAplZdoPoll(void);
```

Description

This function can be used by an End Device to poll its parent for pending data.

Since an End Device is able to sleep, messages addressed to the End Device are buffered by the parent for delivery when the child is ready. This function requests this buffered data and should normally be called immediately after waking from sleep.

This function call will trigger a confirmation event, ZPS_EVENT_NWK_POLL_CONFIRM, if the poll request is successfully sent to the parent. The subsequent arrival of data from the parent is indicated by a ZPS_EVENT_APS_DATA_INDICATION event. Any messages forwarded from the parent should then be collected using the function **ZQ_bZQueueReceive()**.

Parameters

None

Returns

ZPS_E_SUCCESS

APS return codes, listed and described in Section 10.2.2

NWK return codes, listed and described in Section 10.2.3

MAC return codes, listed and described in Section 10.2.4

ZPS_eAplZdoLeaveNetwork

```
ZPS_teStatus ZPS_eAplZdoLeaveNetwork(
                                    uint64 u64Addr,
                                    bool bRemoveChildren,
                                    bool bRejoin);
```

Description

This function can be used to request a node to leave the network. The leaving node can be a child of the requesting node or can be the requesting node itself (excluding the Co-ordinator).

The node being asked to leave the network is specified by means of its IEEE (MAC) address (or zero, if a node is requesting itself to leave the network). You must also:

- Use the parameter *bRemoveChildren* to specify whether children of the leaving node must leave their parent - if this is the case, the leaving node will automatically call **ZPS_eAplZdoLeaveNetwork()** for each of its children. This parameter must always be set to FALSE when the function is called on an End Device (as there are no children).
- Use the parameter *bRejoin* to specify whether the leaving node must attempt to rejoin the network (probably via another parent) immediately after leaving.

> **Tip:** If you wish to move a whole network branch from under the requesting node to a different parent node, set *bRemoveChildren* to FALSE and *bRejoin* to TRUE.

If this function successfully initiates the removal of a node, ZPS_E_SUCCESS will be returned. Subsequently, when the removal is complete, the stack event ZPS_EVENT_NWK_LEAVE_CONFIRM is generated. For details of this event, refer to Section 7.2.2.12.

Parameters

u64Addr	64-bit IEEE (MAC) address of node to leave network (zero value will cause requesting node to leave network)
bRemoveChildren	Boolean value indicating whether children of leaving node must leave their parent: TRUE: Children to leave FALSE: Children not to leave
bRejoin	Boolean value indicating whether leaving node must attempt to rejoin network immediately after leaving: TRUE: Rejoin network immediately FALSE: Do not rejoin network

Returns

ZPS_E_SUCCESS (removal of node successfully started)

APS return codes, listed and described in Section 10.2.2

NWK return codes, listed and described in Section 10.2.3

MAC return codes, listed and described in Section 10.2.4

ZPS_vNwkNibSetLeaveAllowed

```
void ZPS_vNwkNibSetLeaveAllowed(void *pvNwk,
                                bool bLeave);
```

Description

This function can be called on a Router or End Device to determine whether the device should leave the network on receiving a leave request. It has no effect on a Co-ordinator.

- If called with *bLeave* set to TRUE, the device will obey a leave request.
- If called with *bLeave* set to FALSE, the device will ignore a leave request.

Parameters

pvNwk Pointer to NWK layer instance

bLeave Boolean value indicating whether the device will leave the network when requested or will ignore leave request messages:
TRUE - Obey leave request messages
FALSE - Ignore leave request messages

Returns

None

ZPS_vNwkNibSetLeaveRejoin

```
void ZPS_vNwkNibSetLeaveRejoin(void *pvNwk,
                               bool bRejoin);
```

Description

This function can be called on a Router or End Device to configure the device to automatically rejoin after leaving the network, even when a 'leave without rejoin' was requested.

- If called with *bRejoin* set to TRUE, the device will rejoin following a leave.
- If called with *bRejoin* set to FALSE, the device will not rejoin following a leave.

Parameters

pvNwk Pointer to NWK layer instance

bLeave Boolean value indicating whether the device will rejoin the network following a leave:
TRUE - Rejoin the network
FALSE - Do not rejoin the network

Returns

None

ZPS_vSetTablesClearOnLeaveWithoutRejoin

```
void ZPS_vSetTablesClearOnLeaveWithoutRejoin(
                                        bool_t bClear);
```

Description

This function can be called on a Router or End Device to configure whether various tabulated context data must be cleared from the node when it leaves the network without the intention to rejoin.

By default, the Neighbour table, Binding table and Group table are cleared on a Router, and the network key is cleared on a Router and End Device. In addition, other devices remove the node from their Binding tables on detecting the leave request (without the rejoin flag set).

This function can be used to over-ride this behaviour in order to preserve this table data. It can also be used to later reinstate the default behaviour.

Parameters

bClear	Boolean value indicating whether the node should clear the table data when leaving the network without a future rejoin: TRUE - Clear table data (default behaviour) FALSE - Do not clear table data

Returns

None

ZPS_vNtSetUsedStatus

```
void ZPS_vNtSetUsedStatus(
        void *pvNwk,
        ZPS_tsNwkActvNtEntry *psActvNtEntry,
        bool_t bStatus);
```

Description

This function can be used to set the status of a local Neighbour Table to either ‘used’ or ‘unused’:

- Setting the status of an entry to unused effectively removes the entry from the table.
- Setting the status of an entry to used effectively adds an entry to the table.

When adding an entry to the table, it is first necessary for the application to find an entry marked unused. The entry can then be populated with data and marked as used via this function.

Parameters

pvNwk	Pointer to NWK layer instance
psActvNtEntry	Pointer to Neighbour Table entry to access (this must be populated with data when adding a new entry to the table)
bStatus	Entry status to be set: TRUE - Set entry status to ‘used’ FALSE - Set entry status to ‘unused’

Returns

None

ZPS_vNwkSendNwkStatusCommand

```
void ZPS_vNwkSendNwkStatusCommand(
                          void *pvNwk,
                          uint16 u16DstAddress,
                          uint16 u16TargetAddress,
                          uint8 u8CommandId,
                          uint8 u8Radius);
```

Description

This function can be used to send a network status command to another node. For example, it can be used by an End Device to report a routing problem (concerning a remote node) to its parent.

Parameters

pvNwk	Pointer to NWK layer instance
u16DstAddress	Network address of the remote node to which the status command relates (e.g. the node for which a routing problem is being reported)
u16TargetAddress	Network address of the node to which the status command is to be sent (e.g. the parent of the local node)
u8CommandId	Value representing the network status command to be sent (the possible values are provided in the ZigBee PRO specification)
u8Radius	Maximum number of hops permitted to target node (zero value specifies that default maximum is to be used)

Returns

None

ZPS_eAplZdoRegisterZdoLeaveActionCallback

```
void ZPS_eAplZdoRegisterZdoLeaveActionCallback(
                                              void *fnPtr);
```

Description

This function can be used to register a user-defined callback function that will be invoked when a leave request, a management leave request or a remove device request (from a remote node, normally the Trust Centre) is received by the local node. The callback function must determine whether the request must be obeyed or ignored by the stack - this decision may depend on the originator of the request.

The prototype of the callback function is as follows:

bool_t ZPS_bPerformLeaveActionDecider(uint8 *u8Value*,
uint64 *u64Address*,
uint8 *u8Flags***);**

where:

- *u8Value* is an enumerated value indicating the type of request - one of:
  ZPS_LEAVE_ORIGIN_NLME (NLME-LEAVE.request from NWK layer)
  ZPS_LEAVE_ORIGIN_MGMT_LEAVE (management leave request)
  ZPS_LEAVE_ORIGIN_REMOVE_DEVICE (remove request from remote node)
- *u64Address* is the IEEE/MAC address of the node that issued the request
- *u8Flags* is a user-defined bitmap containing flagged information

The callback function must return TRUE to allow or FALSE to disallow the requested leave.

Parameters

fnPtr	Pointer to user-defined callback function to be registered

Returns

None

6.1.2 Security Functions

The ZDO Security functions are used to set up network security (at the 'standard' level), including the keys used in the encryption/decryption of network communications.

The functions are listed below, along with their page references:

Note 1: Before using the above functions on a node, security must be enabled on the node via the device parameter *Security Enabled* in the ZPS Configuration Editor (security is enabled by default).

Note 2: Enabling security also enables many-to-one routing towards the Trust Centre, which will become a network concentrator. You must set the maximum number of nodes to be serviced by the Trust Centre using its network parameter *Route Record Table Size* in the ZPS Configuration Editor (the default number is 4).

Note 3: Many of the security settings and keys that are set up using the above functions can alternatively be pre-configured via the ZPS Configuration Editor.

ZPS_vAplSecSetInitialSecurityState

```
ZPS_teStatus ZPS_vAplSecSetInitialSecurityState(
                        ZPS_teZdoNwkKeyState eState,
                        uint8 *pu8Key,
                        uint8 u8KeySeqNum
                        ZPS_teApsLinkKeyType eKeyType);
```

Description

This function is used to configure the initial state of ZigBee security on the local node. This requires a security key to be specified that will be used in setting up network-level security. Note that before using this function, security must be enabled on the node via the device parameter *Security Enabled* in the ZPS Configuration Editor.

You must provide a pointer to an initial link key of one of the following types:

- Pre-configured global link key
- Pre-configured unique link key

These key types are described in Section 5.8.2. The network key that is randomly generated by the Trust Centre is communicated to the local node in encrypted form using the specified link key.

Parameters

eState	The state of the link key, one of: ZPS_ZDO_PRECONFIGURED_LINK_KEY ZPS_ZDO_ZLL_LINK_KEY
pu8Key	Pointer to pre-configured link key
u8KeySeqNum	Not used when specifying a link key - ignore this parameter
eKeyType	Type of link key, one of: ZPS_APS_UNIQUE_LINK_KEY ZPS_APS_GLOBAL_LINK_KEY

Returns

ZPS_E_SUCCESS (security state successfully initialised)

APS return codes, listed and described in Section 10.2.2

NWK return codes, listed and described in Section 10.2.3

MAC return codes, listed and described in Section 10.2.4

ZPS_eAplZdoTransportNwkKey

```
ZPS_teStatus ZPS_eAplZdoTransportNwkKey(
                    uint8 u8DstAddrMode,
                    ZPS_tuAddress uDstAddress,
                    uint8 au8Key[ZPS_SEC_KEY_LENGTH],
                    uint8 u8KeySeqNum,
                    bool bUseParent,
                    uint64 u64ParentAddr);
```

Description

This function can be used on the Trust Centre to send the network key to one or multiple nodes. On reaching the target node(s), the key is only stored but can be subsequently designated the active network key using the function **ZPS_eAplZdoSwitchKeyReq()**.

The target node can be specified by means of its network address or IEEE/MAC address. A broadcast to multiple nodes in the network can be achieved by specifying a special network address or IEEE/MAC address - see Section 8.3.

If the destination is a single node, it is possible to send the key to the parent of the destination node.

Note that this function will also reset the frame counter on the target node(s).

Parameters

u8DstAddrMode	Type of destination address: ZPS_E_ADDR_MODE_SHORT - 16-bit network address ZPS_E_ADDR_MODE_IEEE - 64-bit IEEE/MAC address All other values are reserved
uDstAddress	Destination address (address type as specified through *u8DstAddrMode*) - special broadcast addresses are detailed in Section 8.3
au8Key[]	Array containing the network key to be transported. This array has a length equal to ZPS_SEC_KEY_LENGTH
u8KeySeqNum	Sequence number of the specified key
bUseParent	Indicates whether to send key to parent of target node: TRUE - send to parent FALSE - do not send to parent
u64ParentAddr	64-bit IEEE/MAC address of parent (if used)

Returns

ZPS_E_SUCCESS (key successfully sent)

APS return codes, listed and described in Section 10.2.2

NWK return codes, listed and described in Section 10.2.3

MAC return codes, listed and described in Section 10.2.4

ZPS_eAplZdoSwitchKeyReq

```
ZPS_teStatus ZPS_eAplZdoSwitchKeyReq(
                    uint8 u8DstAddrMode,
                    ZPS_tuAddress uDstAddress,
                    uint8 u8KeySeqNum);
```

Description

This function can be used (normally by the Trust Centre) to request one or multiple nodes to switch to a different active network key. The new network key is specified using its unique sequence number and the key must have been pre-loaded into the target node(s) using the function **ZPS_eAplZdoTransportNwkKey()** or **ZPS_eAplZdoRequestKeyReq()**.

The target node can be specified by means of its network address or IEEE/MAC address. A broadcast to multiple nodes in the network can be achieved by specifying a special network address or IEEE/MAC address - see Section 8.3.

Parameters

u8DstAddrMode	Type of destination address: ZPS_E_ADDR_MODE_SHORT - 16-bit network address ZPS_E_ADDR_MODE_IEEE - 64-bit IEEE/MAC address All other values are reserved
uDstAddress	Destination address (address type as specified through *u8DstAddrMode*) - special broadcast addresses are detailed in Section 8.3
u8KeySeqNum	Sequence number of new network key to adopt

Returns

ZPS_E_SUCCESS (request successfully sent)

APS return codes, listed and described in Section 10.2.2

NWK return codes, listed and described in Section 10.2.3

MAC return codes, listed and described in Section 10.2.4

ZPS_eAplZdoRequestKeyReq

```
ZPS_teStatus ZPS_eAplZdoRequestKeyReq(
                        uint8 u8KeyType,
                        uint64 u64IeeePartnerAddr);
```

Description

This function can be used to request a link key from the Trust Centre for application-level security. The possible key types that can be requested are:

- **Application link key:** This key will be used to encrypt/decrypt communications with another 'partner node'. The IEEE/MAC address of this partner node must be specified as part of the function call. The Trust Centre will respond by sending the application link key to both the local node and the partner node. When it arrives, this key will be automatically saved by the stack and the event ZPS_EVENT_ZDO_LINK_KEY will be generated once the link key has been installed and is ready to be used.
- **Trust Centre Link Key (TCLK):** This key will be used to encrypt/decrypt communications between the Trust Centre and the local node. The Trust Centre will respond by sending the TCLK to the requesting node.

In the case of requesting a TCLK, the function parameter *u64IeeePartnerAddr* is ignored.

For more information on requesting link keys, refer to Section 5.8.3.2.

Parameters

u8KeyType	Type of key to request: 2 - application link key 4 - Trust Centre Link Key (TCLK) All other values reserved
u64IeeePartnerAddr	IEEE/MAC address of partner node (for application link key)

Returns

ZPS_E_SUCCESS (request successfully sent)

APS return codes, listed and described in Section 10.2.2

NWK return codes, listed and described in Section 10.2.3

MAC return codes, listed and described in Section 10.2.4

ZPS_eAplZdoAddReplaceLinkKey

```
ZPS_teStatus ZPS_eAplZdoAddReplaceLinkKey(
                    uint64 u64IeeeAddr,
                    uint8 au8Key[ZPS_SEC_KEY_LENGTH],
                    ZPS_teApsLinkKeyType eKeyType);
```

Description

This function can be used to introduce or replace the application link key on the local node, where this key will be used to encrypt and decrypt communications with the specified 'partner node'. If an application link key already exists then it will be replaced.

The function must be called on both the local node and the partner node. Note that the Trust Centre's record of the application link key for this pair of nodes remains unchanged.

If the JCU Non-Volatile Memory Manager (NVM) module is enabled, this function will also save the application link key to Non-Volatile Memory. This allows the key to be automatically recovered during a subsequent cold start (e.g. following a power failure).

The *eKeyType* parameter of this function can be used to specify 'unique' or 'global'. This does not relate to the type of key being added or replaced, which is always a unique key. Setting this parameter to 'unique' means that the node will only ever use the unique key. Setting the parameter to 'global' means that the node will use the unique key, where appropriate, and also the pre-configured global link key, where appropriate - for example, the global key would be used when another node joins the network via the local node.

Parameters

u64IeeeAddr	64-bit IEEE/MAC address of partner node for which the specified link key is valid
au8Key[]	Array containing the link key to be added/replaced. This array has a length equal to ZPS_SEC_KEY_LENGTH
eKeyType	Type of the key to be used by the node (see above), one of: ZPS_APS_UNIQUE_LINK_KEY ZPS_APS_GLOBAL_LINK_KEY

Returns

ZPS_E_SUCCESS (link key successfully installed)

APS return codes, listed and described in Section 10.2.2

NWK return codes, listed and described in Section 10.2.3

MAC return codes, listed and described in Section 10.2.4

ZPS_eAplZdoAddReplaceInstallCodes

```
ZPS_teStatus ZPS_eAplZdoAddReplaceInstallCodes(
        uint64 u64IeeeAddr,
        uint8 au8InstallCode[ZPS_INSTALL_CODE_LENGTH],
        uint8 u8InstallCodeSize,
        ZPS_teApsLinkKeyType eKeyType);
```

Description

This function can be used on the Trust Centre to generate a pre-configured unique link key from an install code, where this key will be used to encrypt and decrypt communications between the Trust Centre and the specified node (install codes are described in the *ZigBee Devices User Guide (JN-UG-3131)*). If a pre-configured link key already exists for the node then it will be replaced.

The function must be called on the Trust Centre only. The other node will have the relevant pre-configured unique link key factory-installed.

If the JCU Non-Volatile Memory Manager (NVM) module is enabled, this function will also save the link key to Non-Volatile Memory. This allows the key to be automatically recovered during a subsequent cold start (e.g. following a power failure).

The *eKeyType* parameter of this function can be used to specify 'unique' or 'global'. This does not relate to the type of key being added or replaced, which is always a unique key. Setting this parameter to 'unique' means that the Trust Centre will only ever use the unique key with this node. Setting the parameter to 'global' means that the Trust Centre will use the pre-configured global link key (if available) when there is no unique link key for the node.

Parameters

u64IeeeAddr	64-bit IEEE/MAC address of node for which the generated link key will be valid
au8InstallCode[]	Array containing the install code - the array length ZPS_INSTALL_CODE_LENGTH is given below in *u8InstallCodeSize*
u8InstallCodeSize	Number of characters in the install code - this is the size of the array *au8InstallCode[]*
eKeyType	Type of the key to be used by the node (see above), one of: ZPS_APS_UNIQUE_LINK_KEY ZPS_APS_GLOBAL_LINK_KEY

Returns

ZPS_E_SUCCESS (link key successfully installed)

APS return codes, listed and described in Section 10.2.2

NWK return codes, listed and described in Section 10.2.3

MAC return codes, listed and described in Section 10.2.4

ZPS_eAplZdoRemoveLinkKey

```
ZPS_teStatus ZPS_eAplZdoRemoveLinkKey(
                              uint64 u64IeeeAddr);
```

Description

This function can be used to remove the current application link key that is used to encrypt and decrypt communications between the local node and the specified ‘partner node’.

The function must be called on both the local node and the partner node. Note that the Trust Centre’s record of the application link key for this pair of nodes remains unchanged.

In the absence of an application link key, communications between these nodes will subsequently be secured using the network key.

Parameters

u64IeeeAddr	64-bit IEEE/MAC address of partner node for which the link key is to be removed

Returns

ZPS_E_SUCCESS (link key successfully removed)

APS return codes, listed and described in Section 10.2.2

NWK return codes, listed and described in Section 10.2.3

MAC return codes, listed and described in Section 10.2.4

ZPS_eAplZdoRemoveDeviceReq

```
ZPS_teStatus ZPS_eAplZdoRemoveDeviceReq(
                                uint64 u64ParentAddr,
                                uint64 u64ChildAddr);
```

Description

This function can be used (normally by the Co-ordinator/Trust Centre) to request another node (such as a Router) to remove one of its children from the network (for example, if the child node does not satisfy security requirements).

The Router receiving this request will ignore the request unless it has originated from the Trust Centre or is a request to remove itself. If the request was sent without APS layer encryption, the device will ignore the request. If APS layer security is not in use, the alternative function **ZPS_eAplZdoLeaveNetwork()** should be used.

Parameters

u64ParentAddr	64-bit IEEE/MAC address of parent to be instructed
u64ChildAddr	64-bit IEEE/MAC address of child node to be removed

Returns

ZPS_E_SUCCESS (request successfully sent)

APS return codes, listed and described in Section 10.2.2

NWK return codes, listed and described in Section 10.2.3

MAC return codes, listed and described in Section 10.2.4

ZPS_eAplZdoSetDevicePermission

```
void ZPS_eAplZdoSetDevicePermission(
        ZPS_teDevicePermissions u8DevicePermissions);
```

Description

This function can be used on any device to set the permissions for certain requests from other nodes. The possible settings are:

- Allow all requests from all other nodes (ALL_PERMITED)
- Do not allow join requests from all other nodes (JOIN_DISALLOWED)
- Do not allow data requests from all other nodes (DATA_REQUEST_DISALLOWED)

The function is particularly useful in disabling the generation of APS (end-to-end) acknowledgements, using DATA_REQUEST_DISALLOWED.

Parameters

u8DevicePermissions	Bitmap of permissions to be set, constructed using the following enumerations: ZPS_DEVICE_PERMISSIONS_ALL_PERMITED ZPS_DEVICE_PERMISSIONS_JOIN_DISALLOWED ZPS_DEVICE_PERMISSIONS_DATA_REQUEST_DISALLOWED

Returns

ZPS_E_SUCCESS (permissions successfully set)

APS return codes, listed and described in Section 10.2.2

NWK return codes, listed and described in Section 10.2.3

MAC return codes, listed and described in Section 10.2.4

ZPS_bAplZdoTrustCenterSetDevicePermissions

ZPS_teStatus
ZPS_bAplZdoTrustCenterSetDevicePermissions(
uint64 *u64DeviceAddr*,
ZPS_teTCDevicePermissions *u8DevicePermissions***);**

Description

This function can be used by the Trust Centre to set the permissions for certain requests from a particular node. The possible settings are:

- Allow all requests from the specified node (ALL_PERMITED)
- Do not allow join requests from the specified node (JOIN_DISALLOWED)
- Do not allow data requests from the specified node (DATA_REQUEST_DISALLOWED)

Parameters

u64DeviceAddr	64-bit IEEE/MAC address of node for which permissions are to be set
u8DevicePermissions	Bitmap of permissions to be set, constructed using the following enumerations: ZPS_TRUST_CENTER_ALL_PERMITED ZPS_TRUST_CENTER_JOIN_DISALLOWED ZPS_TRUST_CENTER_DATA_REQUEST_DISALLOWED

Returns

ZPS_E_SUCCESS (permissions successfully set)

APS return codes, listed and described in Section 10.2.2

NWK return codes, listed and described in Section 10.2.3

MAC return codes, listed and described in Section 10.2.4

ZPS_bAplZdoTrustCenterGetDevicePermissions

```
ZPS_teStatus
ZPS_bAplZdoTrustCenterGetDevicePermissions(
    uint64 u64DeviceAddr,
    ZPS_teTCDevicePermissions *pu8DevicePermissions);
```

Description

This function can be used by the Trust Centre to obtain its own permissions for certain requests from a particular node. The possible settings are:

- Allow all requests from the specified node
- Do not allow join requests from the specified node
- Do not allow data requests from the specified node

Parameters

u64DeviceAddr	64-bit IEEE/MAC address of node for which permissions are to be obtained
pu8DevicePermissions	Pointer to bitmap containing permissions obtained, where: 0 indicates all requests allowed 1 indicates join requests disallowed 2 indicates data requests disallowed 3 indicates data and join requests disallowed Higher bits are reserved for future use

Returns

ZPS_E_SUCCESS (permissions successfully obtained)

APS return codes, listed and described in Section 10.2.2

NWK return codes, listed and described in Section 10.2.3

MAC return codes, listed and described in Section 10.2.4

ZPS_bAplZdoTrustCenterRemoveDevice

```
ZPS_teStatus ZPS_bAplZdoTrustCenterRemoveDevice(
                                uint64 u64DeviceAddr);
```

Description

This function can be used by the Trust Centre to delete a node in its information base.

Parameters

u64DeviceAddr	64-bit IEEE/MAC address of node to be removed from list

Returns

ZPS_E_SUCCESS (node successfully removed from list)

APS return codes, listed and described in Section 10.2.2

NWK return codes, listed and described in Section 10.2.3

MAC return codes, listed and described in Section 10.2.4

ZPS_vTcInitFlash

```
void ZPS_vTcInitFlash(
                ZPS_tsAfFlashInfoSet *psFlashInfoSet,
                ZPS_TclkDescriptorEntry *psTclkStruct);
```

Description

This function can be used on the network Co-ordinator/Trust Centre to enable the persistent storage of the Trust Centre Link Keys (TCLKs) for all nodes in the network.

Each of these keys is a unique application-level link key for a node. The key is used to encrypt/decrypt communications between Trust Centre and the node during the commissioning of the node into the network (and is issued by the Trust Centre to replace the pre-configured unique link key).

The function allows these link keys to be stored in devices Flash memory.

- Information about the Flash memory sector to be used to store the link keys is specified in a `ZPS_tsAfFlashInfoSet` structure.
- Information about an individual link key is stored in RAM in a read-only `ZPS_TclkDescriptorEntry` structure, which is for internal use by the stack. An array of these structures must be allocated in RAM, with one element for each node in the network - for example, if there will be up to 250 nodes in the network, the required allocation is:

```
ZPS_TclkDescriptorEntry sData[250];
```

The application can determine at any time whether this feature is enabled by reading the Boolean variable `bSetTclkFlashFeature`, which reads as TRUE if the feature is enabled and as FALSE if it is disabled.

When a new Trust Centre Link Key has been negotiated for a node, the stack on the Trust Centre notifies the application by means of a ZPS_EVENT_TC_STATUS event. The application can discover the IEEE/MAC address of the corresponding node by calling **ZPS_u64GetFlashMappedIeeeAddress()** with the value of `u16ExtAddrLkup` from the key descriptor passed in the event (when the key table is held in RAM, **ZPS_u64NwkNibGetMappedIeeeAddr()** would be called instead).

Parameters

psFlashInfoSet	Pointer to a structure containing information about the Flash memory sector to be used to store the link keys for the network nodes (see Section 7.2.3.8)
psTclkStruct	Pointer to a structure in RAM that will be used to hold information about the storage of one link key in Flash memory (see Section 7.2.3.9)

Returns

None

ZPS_vSetTCLockDownOverride

```
void ZPS_vSetTCLockDownOverride(
                    void* pvApl,
                    bool_t u8RemoteOverride,
                    bool_t bDisableAuthentications);
```

Description

This function can be called on the network Co-ordinator to disable Trust Centre functionality on the device.

The function provides two configuration options:

- Allow remote devices to over-ride the Trust Centre policy
- Disable authentication of network joins (any transport key will also be disabled)

Parameters

pvApl	Handle for the relevant Application layer instance
u8RemoteOverride	Boolean specifying whether remote over-rides of Trust Centre policy are to be permitted: TRUE - Do not allow remote over-rides; stack will not allow the permit join remotely sent to change it's local state FALSE - Allow remote over-rides; stack will accept permit join requests coming in and obey them
bDisableAuthentications	Boolean specifying whether network join authentications are to be disabled: TRUE - Disable authentications FALSE - Do not disable authentications When this flag is set to TRUE permit join will not be accepted remotely and the TC will not transport a key to any joiner

Returns

None

ZPS_psGetActiveKey

ZPS_tsAplApsKeyDescriptorEntry *ZPS_psGetActiveKey(
uint64 *u64IeeeAddress***,**
uint32* *pu32Index***);**

Description

This function can be used on the Trust Centre to obtain the Pre-configured Unique Link Key for the node with the specified IEEE/MAC address. The function searches the local Key Descriptor Table for an entry corresponding to the specified address. If it finds a relevant entry, it returns the entry as well as the index number of the entry in the table. The required key is in the returned table entry.

Parameters

u64IeeeAddress	IEEE/MAC address of the node of interest
pu32Index	Pointer to a location to receive the index number of the relevant Key Descriptor Table entry

Returns

Pointer to requested Key Descriptor Table entry (for structure, see Section 7.2.3.6)

ZPS_vTCSetCallback

void ZPS_vTCSetCallback(void ****pCallbackFn*);**

Description

This function can be used to register a user-defined callback function on the Trust Centre, where this callback function allows the application to react to a notification from another network node - for example, to decide whether to permit a node to join that may or may not be known to the Trust Centre application.

The prototype of the user-defined callback function is:

bool **bTransportKeyDecider** (**uint16** *u16ShortAddress*,
uint64 *u64DeviceAddress*,
uint64 *u64ParentAddress*,
uint8 *u8Status*,
uint16 *u16Interface*);

where:

- *u16ShortAddr* is the network address of the relevant node
- *u64DeviceAddress* is the IEEE/MAC address of the relevant node
- *u64ParentAddress* is the IEEE/MAC address of the parent that sent the notification
- *u8Status* is the nature of the notification:
  0: Secure rejoin
  1: Unsecure join (association)
  2: Leave
  3: Unsecure rejoin
  4: Leave with a rejoin
- *u16Interface* is the MAC interface this join has happened on. If it is 2.4 G only the value is always 0. If it is a MultiMAC device 2.4 G interface, it will return value 0 and sub Gig will return value 1.

To disallow the notified action (e.g. a join), the callback function should return FALSE.

If the callback function is not registered or returns TRUE, the Trust Centre will allow the notified action. In the case of a join, the Trust Centre will send the network key in a 'transport key' command to the node, either:

- encrypted with the node's pre-configured link key, if this key is known to the Trust Centre, or
- encrypted with the Trust Centre's default pre-configured link key otherwise (in this case, the joining node will only be able to decrypt the 'transport key' command and complete the join if it also has the Trust Centre's default pre-configured link key)

Registration of this callback function may be useful in controlling rejoins. A node can initially join a network using its pre-configured link key (which is also known by the Trust Centre), but this key may subsequently be replaced on the Trust Centre by an application link key (shared only by the node and the Trust Centre). If the node later leaves the network and loses its context data (including the application link key), it may attempt to rejoin the network using its pre-configured link key again. The callback function can allow the application to decide whether to permit such a rejoin. If the rejoin is to be allowed, the callback function must replace the stored application link key with the pre-configured link key on the Trust Centre before returning TRUE.

Parameters

pCallbackFn	Pointer to user-defined callback function

Returns

None

6.1.3 Addressing Functions

The ZDO Addressing functions allow node addresses to be stored and obtained. They include the group address functions that allow a group of nodes/endpoints, with an assigned group address, to be created and modified (this group can be used as the destinations for a multicast message).

The functions are listed below, along with their page references:

Note: Further addressing functions are provided in the ZDP API and are described in Section 8.1.1.

ZPS_u16AplZdoGetNwkAddr

```
uint16 ZPS_u16AplZdoGetNwkAddr(void);
```

Description

This function obtains the 16-bit network address of the local node.

Parameters

None

Returns

16-bit network address obtained

ZPS_u64AplZdoGetIeeeAddr

```
uint64 ZPS_u64AplZdoGetIeeeAddr(void);
```

Description

This function obtains the 64-bit IEEE (MAC) address of the local node.

Parameters

None

Returns

64-bit IEEE/MAC address obtained

ZPS_eAplZdoAddAddrMapEntry

```
ZPS_teStatus ZPS_eAplZdoAddAddrMapEntry(
                                        uint16 u16NwkAddr,
                                        uint64 u64ExtAddr);
```

Description

This function can be used to add the addresses of a remote node to the local Address Map table. Each entry in this table stores a remote node's 16-bit network address and an index to its 64-bit IEEE (MAC) address in the MAC Address table (see Section 2.2.4). Thus, the function adds the IEEE address to the MAC Address table and then the index of this entry to the Address Map table.

> ***Caution:*** *You should only modify to the Address Map table using the supplied API functions and never write to it directly.*

Parameters

u16NwkAddr	16-bit network address of node to be added
u64ExtAddr	64-bit IEEE/MAC address of node to be added

Returns

ZPS_E_SUCCESS (addresses successfully added to tables)

APS return codes, listed and described in Section 10.2.2

NWK return codes, listed and described in Section 10.2.3

MAC return codes, listed and described in Section 10.2.4

ZPS_vPurgeAddressMap

```
void ZPS_vPurgeAddressMap(void);
```

Description

This function removes all entries from the Address Map table on the local node.

Caution: *You should only modify to the Address Map table using the supplied API functions and never write to it directly.*

Parameters

None

Returns

None

ZPS_u16AplZdoLookupAddr

uint16 ZPS_u16AplZdoLookupAddr(uint64 *u64ExtAddr***);**

Description

This function can be used to search the local Address Map table for the 16-bit network address of the node with a given 64-bit IEEE (MAC) address.

Parameters

u64ExtAddr	64-bit IEEE/MAC address of node to be search for

Returns

16-bit network address obtained

ZPS_u64AplZdoLookupIeeeAddr

```
uint64 ZPS_u64AplZdoLookupIeeeAddr(
                              uint16 u16NwkAddr);
```

Description

This function can be used to search the local Address Map table for the 64-bit IEEE (MAC) address of the node with a given 16-bit network address.

Parameters

u16NwkAddr	16-bit network address of node to be search for

Returns

64-bit IEEE/MAC address obtained

ZPS_u64NwkNibGetMappedIeeeAddr

```
uint64 ZPS_u64NwkNibGetMappedIeeeAddr(
                                      void *pvNwk,
                                      uint16 u16Location);
```

Description

This function can be used to obtain the 64-bit IEEE (MAC) address that is stored in a particular entry in the local MAC Address table. The number of the entry must be specified as well as the handle of the relevant network.

Parameters

pvNwk	Pointer to relevant NWK layer instance
u16Location	Number of entry to access in MAC Address table

Returns

64-bit IEEE/MAC address obtained

ZPS_u64GetFlashMappedIeeeAddress

```
uint64 ZPS_u64GetFlashMappedIeeeAddress(
                                uint16 u16Location);
```

Description

This function can be used on the Trust Centre to obtain the 64-bit IEEE (MAC) address of the node for which a link key has been persistently stored in the specified location in devices Flash memory. The location is specified as the number of the array element for the node - see the description of **ZPS_vTcInitFlash()** on page 108.

Parameters

u16Location Number of the array element for the node

Returns

64-bit IEEE/MAC address obtained

ZPS_bNwkFindAddIeeeAddr

```
bool_t ZPS_bNwkFindAddIeeeAddr(
                    void *pvNwk,
                    uint64 u64IeeeAddr,
                    uint16 *pu16Location,
                    bool_t bNeighborTable;
```

Description

This function can be used to add the 64-bit IEEE (MAC) address of a node to the local MAC Address table. The function will first search the table to determine whether the address already exists in the table. If there is no entry for this address, a new entry for it will be added to the table. The number of the entry where the address was found or added is returned in a specified location.

> ***Caution:** You should only modify to the MAC Address table using this API function and never write to it directly.*

Parameters

pvNwk	Pointer to relevant NWK layer instance
u64IeeeAddr	64-bit IEEE/MAC address to be added
pu16Location	Pointer to location to receive number of entry in MAC Address table where specified address was found or added
bNeighborTable	Always set to FALSE

Returns

Boolean indicating the outcome of the operation:

TRUE - address successfully added to the table

FALSE - address found to already exist in the table

ZPS_vSetOverrideLocalIeeeAddr

```
void ZPS_vSetOverrideLocalIeeeAddr(
                    uint64 *pu64Address);
```

Description

This function can be used to over-ride the 64-bit IEEE (MAC) address of the device where this address is stored locally in the index sector of Flash memory.

Caution: *If required, this function must be called before the ZigBee PRO stack is initialised.*

Parameters

pu64Address	Pointer to the 64-bit IEEE MAC address

Caution: *The stack stores a pointer to pu64Address and does not take a copy of the address. The memory pointed to by pu64Address must therefore be static or constant, and must not be on the CPU stack.*

ZPS_eAplZdoGroupEndpointAdd

```
ZPS_teStatus ZPS_eAplZdoGroupEndpointAdd(
                        uint16 u16GroupAddr,
                        uint8 u8DstEndpoint);
```

Description

This function requests that the specified endpoint (on the local node) is added to the group with the specified group address. This means that this endpoint will become one of the destinations for messages sent to the given group address.

To form a group comprising endpoints from different nodes, it is necessary to call this function for each endpoint individually, on the endpoint's local node.

An endpoint can belong to more than one group.

Information on the endpoints in a group can be obtained from the Group Address table in the AIB (which can be accessed using the function **ZPS_psAplAibGetAib()**).

Note: In order to add an endpoint to a group using this function, a Group Address table must exist on the local node. This table is created using the ZPS Configuration Editor.

Parameters

u16GroupAddr	16-bit group address
u8DstEndpoint	Number of destination endpoint (1-240) on local node

Returns

ZPS_E_SUCCESS (endpoint successfully added to group)

APS return codes, listed and described in Section 10.2.2

NWK return codes, listed and described in Section 10.2.3

MAC return codes, listed and described in Section 10.2.4

ZPS_eAplZdoGroupEndpointRemove

```
ZPS_teStatus ZPS_eAplZdoGroupEndpointRemove(
                                  uint16 u16GroupAddr,
                                  uint8 u8DstEndpoint);
```

Description

This function requests that the specified endpoint (on the local node) is removed from the group with the specified group address.

If you wish to remove an endpoint from all groups to which it belongs, use the function **ZPS_eAplZdoGroupAllEndpointRemove()**.

Information on the endpoints in a group can be obtained from the Group Address table in the AIB (which can be accessed using the function **ZPS_psAplAibGetAib()**).

Parameters

u16GroupAddr	16-bit group address
u8DstEndpoint	Number of destination endpoint (1-240) on local node

Returns

ZPS_E_SUCCESS (endpoint successfully removed from group)

APS return codes, listed and described in Section 10.2.2

NWK return codes, listed and described in Section 10.2.3

MAC return codes, listed and described in Section 10.2.4

ZPS_eAplZdoGroupAllEndpointRemove

```
ZPS_teStatus ZPS_eAplZdoGroupAllEndpointRemove(
                                    uint8 u8DstEndpoint);
```

Description

This function requests that the specified endpoint (on the local node) is removed from all groups to which it currently belongs.

Information on the endpoints in a group can be obtained from the Group Address table in the AIB (which can be accessed using the function **ZPS_psAplAibGetAib()**).

Parameters

u8DstEndpoint	Number of destination endpoint (1-240) on local node

Returns

ZPS_E_SUCCESS (endpoint successfully removed from all groups)

APS return codes, listed and described in Section 10.2.2

NWK return codes, listed and described in Section 10.2.3

MAC return codes, listed and described in Section 10.2.4

6.1.4 Routing Functions

The ZDO Routing functions can be used to make route discovery requests.

The functions are listed below, along with their page references:

ZPS_eAplZdoRouteRequest

ZPS_teStatus ZPS_eAplZdoRouteRequest(
uint16 *u16DstAddr*,
uint8 *u8Radius***);**

Description

This function requests the discovery of a route to the specified remote node (and that this route is added to the Routing tables in the relevant Router nodes).

Parameters

u16DstAddr	16-bit network address of destination node
u8Radius	Maximum number of hops permitted to destination node (zero value specifies that default maximum is to be used)

Returns

ZPS_E_SUCCESS (route discovery request successfully initiated)

APS return codes, listed and described in Section 10.2.2

NWK return codes, listed and described in Section 10.2.3

MAC return codes, listed and described in Section 10.2.4

ZPS_eAplZdoManyToOneRouteRequest

```
ZPS_teStatus ZPS_eAplZdoManyToOneRouteRequest(
                                        bool bCacheRoute,
                                        uint8 u8Radius);
```

Description

This function requests a 'many-to-one' route discovery and should be called on a node that will act as a 'concentrator' in the network (that is, a node with which many other nodes will need to communicate).

As a result of this function call, a route discovery message is broadcast across the network and Routing table entries (for routes back to the concentrator) are stored in the Router nodes.

The maximum number of hops to be taken by a route discovery message in this broadcast must be specified. There is also an option to store the discovered routes in a Route Record Table on the concentrator (for return communications).

Parameters

bCacheRoute	Indicates whether to store routes in Route Record Table: TRUE - store routes FALSE - do not store routes
u8Radius	Maximum number of hops of route discovery message (zero value specifies that default maximum is to be used)

Returns

ZPS_E_SUCCESS (many-to-one route discovery successfully initiated)

APS return codes, listed and described in Section 10.2.2

NWK return codes, listed and described in Section 10.2.3

MAC return codes, listed and described in Section 10.2.4

6.1.5 Object Handle Functions

The ZDO Object Handle functions can be used to obtain the handles of various objects.

The functions are listed below, along with their page references:

ZPS_pvAplZdoGetAplHandle

```
void *ZPS_pvAplZdoGetAplHandle(void);
```

Description

This function obtains a handle for the Application layer instance.

Parameters

None

Returns

Pointer to Application layer instance

ZPS_pvAplZdoGetMacHandle

```
void *ZPS_pvAplZdoGetMacHandle(void);
```

Description

This function obtains a handle for the IEEE 802.15.4 MAC layer instance.

Parameters

None

Returns

Pointer to MAC layer instance

ZPS_pvAplZdoGetNwkHandle

```
void *ZPS_pvAplZdoGetNwkHandle(void);
```

Description

This function obtains a handle for the ZigBee NWK layer instance.

Parameters

None

Returns

Pointer to NWK layer instance

ZPS_psNwkNibGetHandle

ZPS_tsNwkNib *ZPS_psNwkNibGetHandle(void ****pvNwk*);**

Description

This function obtains a handle for the NIB (Network Information Base) corresponding to the specified NWK layer instance.

The function should be called after **ZPS_pvAplZdoGetNwkHandle()**, which is used to obtain a pointer to the NWK layer instance.

The NIB is detailed in the *ZigBee Specification (05347)* from the ZigBee Alliance.

This function is not strictly a ZDO function.

Parameters

pvNwk	Pointer to NWK layer instance

Returns

Pointer to NIB structure

Example

```
void *pvNwk; = ZPS_pvAplZdoGetNwkHandle();
ZPS_tsNwkNib *pNib = ZPS_psNwkNibGetHandle(pvNwk);
```

ZPS_psAplAibGetAib

```
ZPS_tsAplAib *ZPS_psAplAibGetAib(void);
```

Description

This function obtains a pointer to the AIB (Application Information Base) structure for the application.

Parameters

None

Returns

Pointer to AIB structure

ZPS_psAplZdoGetNib

```
ZPS_tsNwkNib *ZPS_psAplZdoGetNib(void);
```

Description

This function obtains a pointer to the NIB (Network Information Base) structure.

The NIB is detailed in the *ZigBee Specification (05347)* from the ZigBee Alliance.

Parameters

None

Returns

Pointer to NIB structure

ZPS_u64NwkNibGetEpid

```
uint64 ZPS_u64NwkNibGetEpid(void *pvNwk);
```

Description

This function can be used to obtain the Extended PAN ID (EPID) from a local NIB (Network Information Base).

The handle of the NWK layer instance that contains the relevant NIB must be specified. This handle can be obtained using **ZPS_pvAplZdoGetNwkHandle()**.

Parameters

pNibHandle	Pointer to NWK layer instance that contains the NIB

Returns

64-bit Extended PAN ID from NIB

6.1.6 Optional Cluster Function

The ZDO Optional Cluster function can be used to register a user-defined callback function to handle messages for a ZDO cluster that is not currently supported by the NXP ZigBee PRO stack.

The function is listed below, along with its page reference:

ZPS_eAplZdoRegisterZdoFilterCallback

```
ZPS_teStatus ZPS_eAplZdoRegisterZdoFilterCallback(
                                        void *fnptr);
```

Description

This function can be used to register a user-defined callback function which handles messages received for an unsupported cluster which resides on the ZDO endpoint (0), such as the cluster for an optional descriptor (e.g. user descriptor).

The prototype of the user-defined callback function is:

bool fn(uint16 *clusterid***);**

where *clusterid* is the ID of the cluster that the function handles.

Normally, a message arriving for an unsupported ZDO cluster is not handled and the stack automatically returns an ‘unsupported’ message to the originating node. If this function is used to register a callback function for an unsupported ZDO cluster then on receiving a message for the cluster, the stack will invoke the callback function. The stack will not respond with an ‘unsupported message’ provided that the callback function returns TRUE, otherwise the normal stack behaviour will continue.

The callback function allows the received message to be passed to the application for servicing.

Parameters

fnptr	Pointer to user-defined callback function

Returns

ZPS_E_SUCCESS (callback function successfully registered)

APS return codes, listed and described in Section 10.2.2

NWK return codes, listed and described in Section 10.2.3

MAC return codes, listed and described in Section 10.2.4

6.2 ZDO Enumerations

This section details the enumerated types used by the ZDO functions. These are all defined in the header file **zps_apl_zdo.h**.

6.2.1 Security Keys (ZPS_teZdoNwkKeyState)

This structure `ZPS_teZdoNwkKeyState` contains the enumerations used to specify a type of security key:

```
typedef enum
{
    ZPS_ZDO_NO_NETWORK_KEY,
    ZPS_ZDO_PRECONFIGURED_LINK_KEY,
    ZPS_ZDO_DISTRIBUTED_LINK_KEY,
    ZPS_ZDO_PRCONFIGURED_INSTALLATION_CODE
} PACK ZPS_teZdoNwkKeyState
```

These enumerations are described in the table below:

Enumeration	Description
ZPS_ZDO_NO_NETWORK_KEY	No network key is to be used.
ZPS_ZDO_PRECONFIGURED_LINK_KEY	A pre-configured link key is to be used. This key may be fixed at the time of manufacture.
ZPS_ZDO_DISTRIBUTED_LINK_KEY	A pre-configured ZigBee Light Link (ZLL) link key is to be used. This key may be fixed at the time of manufacture. A ZLL node will contain both a ZPS_ZDO_PRECONFIG-URED_LINK_KEY for Home Automation (HA) compatibility and a ZPS_ZDO_ZLL_LINK_KEY for ZLL networks.
ZPS_ZDO_PRCONFIGURED_INSTALLA-TION_CODE	A preconfigured install code is to be used. This results in a key being generated from the install code.

Table 4: Security Key Enumerations

6.2.2 Device Types (ZPS_teZdoDeviceType)

This structure `ZPS_teZdoDeviceType` contains the enumerations used to specify a ZigBee device type

```
typedef enum
{
    ZPS_ZDO_DEVICE_COORD,
    ZPS_ZDO_DEVICE_ROUTER,
    ZPS_ZDO_DEVICE_ENDDEVICE
} PACK ZPS_teZdoDeviceType;
```

These enumerations are described in the table below.

Enumeration	Description
ZPS_ZDO_DEVICE_COORD	Co-ordinator
ZPS_ZDO_DEVICE_ROUTER	Router
ZPS_ZDO_DEVICE_ENDDEVICE	End Device

Table 5: Device Type Enumerations

6.2.3 Device Permissions (ZPS_teDevicePermissions)

This structure `ZPS_teDevicePermissions` contains the enumerations used on a device to specify the permissions for certain requests from other nodes:

```
typedef enum
{
    ZPS_DEVICE_PERMISSIONS_ALL_PERMITED = 0,
    ZPS_DEVICE_PERMISSIONS_JOIN_DISALLOWED = 1,
    ZPS_DEVICE_PERMISSIONS_DATA_REQUEST_DISALLOWED = 2,
    ZPS_DEVICE_PERMISSIONS_REJOIN_DISALLOWED = 4,
} PACK ZPS_teDevicePermissions;
```

These enumerations are described in the table below:

Enumeration	Description
ZPS_DEVICE_PERMISSIONS_ALL_PERMITED	Allow all requests from other nodes
ZPS_DEVICE_PERMISSIONS_JOIN_DISALLOWED	Do not allow join requests from other nodes
ZPS_DEVICE_PERMISSIONS_DATA_REQUEST_DISALLOWED	Do not allow data requests from other nodes and disable end-to-end acknowledgements

Table 6: Device Permissions Enumerations

Enumeration	Description
ZPS_DEVICE_PERMISSIONS_REJOIN_DISALLOWED	Do not allow insecure rejoin.

Table 6: Device Permissions Enumerations

7. Application Framework (AF) API

The chapter describes the resources of the Application Framework (AF) API. This API is concerned with transmitting data, controlling/monitoring local endpoints, and copying descriptors to/from the context area of the stack. The API is defined in the header file **zps_apl_af.h**.

In this chapter:

- Section 7.1 details the AF API functions
- Section 7.2 details the AF API structures

7.1 AF API Functions

The AF API functions are divided into the following categories:

- **Initialisation** functions, described in Section 7.1.1
- **Data Transfer** functions, described in Section 7.1.2
- **Endpoint** functions, described in Section 7.1.3
- **Descriptor** functions, described in Section 7.1.4

7.1.1 Initialisation Functions

The AF API contains four initialisation functions.

The functions are listed below, along with their page references:

Note: The function **ZPS_eAplAfInit()** is mandatory and must be the first network function called in your application.

ZPS_eAplAfInit

ZPS_teStatus ZPS_eAplAfInit(void);

Description

This function initialises the Application Framework and must be the first network function called in your application code. The function will first request a reset of the Network (NWK) layer of the ZigBee PRO stack. It will then initialise certain network parameters with values that have been pre-configured using the ZPS Configuration Editor (see Chapter 12). These parameters include the node type and the Extended PAN ID of the network.

Note: This function also resets the IEEE 802.15.4 MAC and PHY levels of the stack. Therefore, if any customised MAC or PHY settings are required, these must be made after this function has been called (such settings could be made with the 802.15.4 Stack API and/or the Integrated Peripherals API, which are supplied in the JN518x or K32W041/K32W061 SDKs).

The device will be started as the pre-configured node type. If this is a Co-ordinator, the Extended PAN ID of the node is set to the pre-configured value. Note that if a zero value has been specified, the Co-ordinator will use its own IEEE/MAC address for the Extended PAN ID.

Parameters

None

Returns

ZPS_E_SUCCESS (AF successfully initialised)

APS return codes, listed and described in Section 10.2.2

NWK return codes, listed and described in Section 10.2.3

MAC return codes, listed and described in Section 10.2.4

ZPS_vAplAfSetMacCapability

```
void ZPS_vAplAfSetMacCapability(uint8 u8MacCapability);
```

Description

This function can be used on a Router or End Device to configure the IEEE 802.15.4 MAC capabilities in the Node descriptor. The MAC capabilities are specified in an 8-bit bitmap, detailed in the table below.

Bits	Description
0	Co-ordinator capability: 1: Node able to act as Co-ordinator 0: Node not able to act as Co-ordinator
1	Device type: 1: Full-Function Device (FFD) 0: Reduced-Function Device (RFD) An FFD can act as any node type while an RFD cannot act as the network Co-ordinator.
2	Power source: 1: Node is mains-powered 0: Node is not mains-powered
3	Receiver on when idle: 1: Receiver enabled during idle periods 0: Receiver disabled during idle periods to conserve power
4-5	Reserved
6	Security capability: 1: High security 0: Standard security
7	Allocate address: 1: Network address should be allocated to node 0: Network address need not be allocated to node

Parameters

u8MacCapability Bitmap containing the MAC capabilities to be configured (see table above)

Returns

None

ZPS_eAplAibSetApsUseExtendedPanId

```
ZPS_teStatus ZPS_eAplAibSetApsUseExtendedPanId(
                                     uint64 u64UseExtPanId);
```

Description

This function can be used to create an application record of the Extended PAN ID (EPID) of the network to which the local device belongs.

- The only use of this function for a Co-ordinator is described in Section 5.1.1.
- The function should only be called on a Router or End Device in the manner described in Section 5.1.2.

Parameters

u64UseExtPanId	Extended PAN ID of network to which device belongs

Returns

ZPS_E_SUCCESS (Extended PAN ID record successfully created)

APS return codes, listed and described in Section 10.2.2

NWK return codes, listed and described in Section 10.2.3

MAC return codes, listed and described in Section 10.2.4

ZPS_vExtendedStatusSetCallback

void ZPS_vExtendedStatusSetCallback(
tpfExtendedStatusCallBack *pfExtendedStatusCallBack***);**

Description

This function can be used to register a callback function for extended error handling (see Section 5.7).

The prototype of the callback function is:

ZPS_teExtendedStatus vExtendedStatusCb();

The registered callback function will be invoked if a subsequent API function call results in one of the following errors:

- 0xA3: ZPS_APL_APS_E_ILLEGAL_REQUEST
- 0xA6: ZPS_APL_APS_E_INVALID_PARAMETER
- 0xC2: ZPS_NWK_ENUM_INVALID_REQUEST

The callback function will return another error code (from those listed and described in Section 10.2.5) which provides a more specific reason for the error.

Parameters

pfExtendedStatusCallBack	Pointer to extended error handling callback function to be registered

Returns

None

ZPS_bAppAddBeaconFilter

```
void ZPS_bAppAddBeaconFilter(
                    tsBeaconFilterType *psAppBeaconStruct);
```

Description

This function can be used to introduce a filter that will be used for filtering beacons in network searches (on a Router or End Device). Beacons can be filtered on the basis of PAN ID, Extended PAN ID, LQI value and device joining status/capacity. The filter details are provided in a `tsBeaconFilterType` structure (see Section 7.2.3.5).

If required, this function should be called immediately before **ZPS_eAplZdoDiscoverNetworks()**, **ZPS_eAplZdoRejoinNetwork()** or **ZPS_eAplZdoStartStack()**.

Caution: *A filter should NOT be implemented unless attempting a join, as this will prevent some stack operations from working correctly.*

Once the join or discovery has completed, the filter is automatically removed and will need to be re-instated if a retry is required.

Guidelines on the implementation of beacon filters are provided in Appendix B.4.

Parameters

**psAppBeaconStruct*	Pointer to a structure containing the beacon filter details (see Section 7.2.3.5)

Returns

None

ZPS_eAplFormDistributedNetworkRouter

ZPS_teStatus ZPS_eAplFormDistributedNetworkRouter(
ZPS_tsAftsStartParamsDistributed ******psStartParms*,
bool_t *bSendDeviceAnnce***);**

Description

This function can be used on a Router node to introduce the node into a distributed security network (see Section 5.10.2). The function must be called on the Router node that creates the distributed security network - therefore, the first node of the new network. Subsequent Router nodes may be introduced using this function, but could be introduced using other commissioning methods, such as Touchlink.

Parameters

psStartParms	Pointer to structure containing the start parameter values for the Router- see Section 7.2.3.7
bSendDeviceAnnce	Boolean indicating whether a device announcement message is to be sent: TRUE - send device announcement FALSE - do not send device announcement

Returns

ZPS_E_SUCCESS (network successfully created)

APS return codes, listed and described in Section 10.2.2

NWK return codes, listed and described in Section 10.2.3

MAC return codes, listed and described in Section 10.2.4

ZPS_eAplInitEndDeviceDistributed

```
ZPS_teStatus ZPS_eAplInitEndDeviceDistributed(
        ZPS_tsAftsStartParamsDistributed *psStartParms);
```

Description

This function can be used on an End Device node to introduce the node into a distributed security network (see Section 5.10.2). This network must have already been created by a Router using the **ZPS_eAplFormDistributedNetworkRouter()** function. End Device nodes may be introduced into the network in this way or using other commissioning methods, such as Touchlink.

Parameters

psStartParms	Pointer to structure containing the start parameter values for the End Device - see Section 7.2.3.7

Returns

ZPS_E_SUCCESS (network successfully created)

APS return codes, listed and described in Section 10.2.2

NWK return codes, listed and described in Section 10.2.3

MAC return codes, listed and described in Section 10.2.4

ZPS_vAplAfEnableMcpsFilter

```
void ZPS_vAplAfEnableMcpsFilter(
                        bool bEnableFilter,
                        uint8 u8LinkCostThreshold);
```

Description

This function allows packet filtering based on ‘link cost’ to be enabled/disabled, as well as some basic configuration of the filtering. Packet filtering is disabled by default.

The default ‘link cost threshold’ is 5. This means that when packet filtering is enabled, received packets with a link cost of 5 or less will be discarded by the stack and not queued for processing. The link cost threshold can be modified (from the default value of 5) using this function.

If required, this function can be called at any time after **ZPS_eAplAfInit()**.

For more information on packet filtering and link costs, refer to Section 5.10.3.

Parameters

bEnableFilter	Enables or disables packet filtering: TRUE - Enable packet filtering FALSE - Disable packet filtering
u8LinkCostThreshold	Link cost threshold to be set (only valid when packet filtering is enabled)

Returns

None

ZPS_vNwkLinkCostCallbackRegister

void ZPS_vNwkLinkCostCallbackRegister(void ****pvFn*);**

Description

This function can be used to register a user-defined callback function which defines custom mappings between LQI values and link costs that are to be used in packet filtering based on link cost. When packet filtering is enabled, the stack uses a default set of mappings, detailed in Section 5.10.3.1. The callback function is only needed if custom mappings are to be used that will over-ride the default mappings. If required, this registration function must be called before **ZPS_eAplAfInit()**, and on both cold and warm starts.

The user-defined callback function to be registered has the following prototype:

uint8 APP_u8LinkCost(uint8 *u8Lqi***);**

This callback function translates a measured LQI value (*u8Lqi*) into a link cost value. An example function is given in Section 5.10.3.3.

For more information on packet filtering and link costs, refer to Section 5.10.3.

Parameters

pvFn	Pointer to user-defined callback function to be registered

Returns

None

7.1.2 Data Transfer Functions

The AF Data Transfer functions are used to request the transmission of data, in the form of an Application Protocol Data Unit (APDU), to one or more remote nodes.

The functions are listed below, along with their page references:

Note: Functions for handling APDUs are provided in the PDUM API, described in the *JN51xx Core Utilities User Guide (JN-UG-3133)*.

APDUs for Requests and Responses

A request generated by this API is sent in an APDU (Application Protocol Data Unit). A local APDU instance for the request must first be allocated using the PDUM function **PDUM_hAPduAllocateAPduInstance()**. This function returns a handle for the APDU instance, which is subsequently used in the relevant AF API request function. Once the request has been successfully sent, the APDU instance is automatically de-allocated by the stack (there is no need for the application to de-allocate it).

Note: If the request is not successfully sent (the send function does not return ZPS_E_SUCCESS) then the APDU instance will not be automatically de-allocated and the application should de-allocate it using the PDUM function **PDUM_eAPduFreeAPduInstance()**.

When a response is subsequently received, the stack automatically allocates a local APDU instance and includes its handle in the notification event for the response. Once the response has been dealt with, the application must de-allocate the APDU instance using the function **PDUM_eAPduFreeAPduInstance()**.

ZPS_eAplAfApsdeDataReq

```
ZPS_teStatus ZPS_eAplAfApsdeDataReq(
                PDUM_thAPduInstance hAPduInst,
                ZPS_tsAfProfileDataReq *psProfileDataReq,
                uint8 *pu8SeqNum);
```

Description

This function submits a request to send data to a remote node, with no restrictions on the type of transmission, destination address, destination application profile, destination cluster and destination endpoint number - these destination parameters do not need to be known to the stack or defined in the ZPS configuration. In this sense, this is most general of the Data Transfer functions.

The destination details and type of transmission are specified in the function call in a `ZPS_tsAfProfileDataReq` structure (see Section 7.2.3.4).

The data is sent in an Application Protocol Data Unit (APDU) instance, which can be allocated using the PDUM function **PDUM_hAPduAllocateAPduInstance()** and then written to using **PDUM_u16APduInstanceWriteNBO()**.

If the APDU size is larger than the maximum packet size allowed on the network, this function call will fail (and return ZPS_E_ADSU_TOO_LONG). To send large APDUs, use the function **ZPS_eAplAfUnicastAckDataReq()**, which automatically implements data fragmentation (if required).

Once the sent data has reached the first hop node in the route to its destination, a ZPS_EVENT_APS_DATA_CONFIRM event will be generated on the local node.

Parameters

hAPduInst	Handle of APDU instance to be sent
**psProfileDataReq*	Pointer to structure containing the details for the transmission (see Section 7.2.3.4)
**pu8SeqNum*	Pointer to location to receive sequence number assigned to data transfer request. If not required, set to NULL

Returns

ZPS_E_SUCCESS

APS return codes, listed and described in Section 10.2.2

NWK return codes, listed and described in Section 10.2.3

MAC return codes, listed and described in Section 10.2.4

ZPS_eAplAfUnicastDataReq

```
ZPS_teStatus ZPS_eAplAfUnicastDataReq(
                    PDUM_thAPduInstance hAPduInst,
                    uint16 u16ClusterId,
                    uint8 u8SrcEndpoint,
                    uint8 u8DstEndpoint,
                    uint16 u16DestAddr,
                    ZPS_teAplAfSecurityMode eSecurityMode,
                    uint8 u8Radius,
                    uint8 *pu8SeqNum);
```

Description

This function submits a request to send data to a remote node (unicast), using the remote node's network address. You must specify the local endpoint and output cluster from which the data originates (the cluster must be in the Simple descriptor for the endpoint), as well as the network address of the remote node and the destination endpoint on the node.

The data is sent in an Application Protocol Data Unit (APDU) instance, which can be allocated using the PDUM function **PDUM_hAPduAllocateAPduInstance()** and then written to using **PDUM_u16APduInstanceWriteNBO()**.

If the APDU size is larger than the maximum packet size allowed on the network, this function call will fail (and return ZPS_E_ADSU_TOO_LONG). To send large APDUs, use the function **ZPS_eAplAfUnicastAckDataReq()**, which automatically implements data fragmentation (if required).

Once the sent data has reached the first hop node in the route to its destination, a ZPS_EVENT_APS_DATA_CONFIRM event will be generated on the local node.

If data is sent using this function to a destination for which a route has not already been established, the data will not be sent and a route discovery will be performed instead. In this case, the function will return ZPS_NWK_ENUM_ROUTE_ERROR and must later be re-called to send the data (see Note under "Unicast" on page 84).

Security (encryption/decryption) can be applied to the APDU, where this security can be implemented at the Application layer or the network (ZigBee) layer, or both.

Parameters

hAPduInst	Handle of APDU instance to be sent
u16ClusterId	Identifier of relevant output cluster on source endpoint
u8SrcEndpoint	Source endpoint number (1-240) on local node
u8DstEndpoint	Destination endpoint number (1-240) on remote node
u16DstAddr	Network address of destination node

eSecurityMode	Security mode for data transfer: ZPS_E_APL_AF_UNSECURE (no security enabled) ZPS_E_APL_AF_SECURE (Application-level security using link key and network key) ZPS_E_APL_AF_SECURE_NWK (Network-level security using network key) ZPS_E_APL_AF_SECURE|ZPS_E_APL_AF_EXT_NONCE (Application-level security using link key and network key with the extended NONCE included in the frame) ZPS_E_APL_AF_WILD_PROFILE (May be combined with above flags using OR operator. Sends the message using the wild card profile (0xFFFF) instead of the profile in the associated Simple descriptor)
u8Radius	Maximum number of hops permitted to destination node (zero value specifies that default maximum is to be used)
**pu8SeqNum*	Pointer to location to receive sequence number assigned to data transfer request. If not required, set to NULL

Returns

ZPS_E_SUCCESS

APS return codes, listed and described in Section 10.2.2

NWK return codes, listed and described in Section 10.2.3

MAC return codes, listed and described in Section 10.2.4

ZPS_eAplAfUnicastIeeeDataReq

```
ZPS_teStatus ZPS_eAplAfUnicastIeeeDataReq(
                    PDUM_thAPduInstance hAPduInst,
                    uint16 u16ClusterId,
                    uint8 u8SrcEndpoint,
                    uint8 u8DstEndpoint,
                    uint64 u64DestAddr,
                    ZPS_teAplAfSecurityMode eSecurityMode,
                    uint8 u8Radius,
                    uint8 *pu8SeqNum);
```

Description

This function submits a request to send data to a remote node (unicast), using the remote node's IEEE (MAC) address. You must specify the local endpoint and output cluster from which the data originates (the cluster must be in the Simple descriptor for the endpoint), as well as the IEEE address of the remote node and the destination endpoint on the node.

The data is sent in an Application Protocol Data Unit (APDU) instance, which can be allocated using the PDUM function **PDUM_hAPduAllocateAPduInstance()** and then written to using **PDUM_u16APduInstanceWriteNBO()**.

If the APDU size is larger than the maximum packet size allowed on the network, this function call will fail (and return ZPS_E_ADSU_TOO_LONG). To send large APDUs, use the function **ZPS_eAplAfUnicastIeeeAckDataReq()**, which automatically implements data fragmentation (if required).

Once the sent data has reached the first hop node in the route to its destination, a ZPS_EVENT_APS_DATA_CONFIRM event will be generated on the local node.

If data is sent using this function to a destination for which a route has not already been established, the data will not be sent and a route discovery will be performed instead. In this case, the function will return ZPS_NWK_ENUM_ROUTE_ERROR and must later be re-called to send the data (see Note under "Unicast" on page 84).

Security (encryption/decryption) can be applied to the APDU, where this security can be implemented at the Application layer or the network (ZigBee) layer, or both.

Parameters

hAPduInst	Handle of APDU instance to be sent
u16ClusterId	Identifier of relevant output cluster on source endpoint
u8SrcEndpoint	Source endpoint number (1-240) on local node
u8DstEndpoint	Destination endpoint number (1-240) on remote node
u64DestAddr	IEEE (MAC) address of destination node

eSecurityMode	Security mode for data transfer: ZPS_E_APL_AF_UNSECURE (no security enabled) ZPS_E_APL_AF_SECURE (Application-level security using link key and network key) ZPS_E_APL_AF_SECURE_NWK (Network-level security using network key) ZPS_E_APL_AF_SECURE|ZPS_E_APL_AF_EXT_NONCE (Application-level security using link key and network key with the extended NONCE included in the frame) ZPS_E_APL_AF_WILD_PROFILE (May be combined with above flags using OR operator. Sends the message using the wild card profile (0xFFFF) instead of the profile in the associated Simple descriptor)
u8Radius	Maximum number of hops permitted to destination node (zero value specifies that default maximum is to be used)
**pu8SeqNum*	Pointer to location to receive sequence number assigned to data transfer request. If not required, set to NULL

Returns

ZPS_E_SUCCESS

APS return codes, listed and described in Section 10.2.2

NWK return codes, listed and described in Section 10.2.3

MAC return codes, listed and described in Section 10.2.4

ZPS_eAplAfUnicastAckDataReq

```
ZPS_teStatus ZPS_eAplAfUnicastAckDataReq(
                    PDUM_thAPduInstance hAPduInst,
                    uint16 u16ClusterId,
                    uint8 u8SrcEndpoint,
                    uint8 u8DstEndpoint,
                    uint16 u16DestAddr,
                    ZPS_teAplAfSecurityMode eSecurityMode,
                    uint8 u8Radius,
                    uint8 *pu8SeqNum);
```

Description

This function submits a request to send data to a remote node (unicast), using the remote node's network address, and requires an acknowledgement to be returned by the remote node once the data reaches its destination. You must specify the local endpoint and output cluster from which the data originates (the cluster must be in the Simple descriptor for the endpoint), as well as the network address of the remote node and the destination endpoint on the node.

The data is sent in an Application Protocol Data Unit (APDU) instance, which can be allocated using the PDUM function **PDUM_hAPduAllocateAPduInstance()** and then written to using **PDUM_u16APduInstanceWriteNBO()**.

If the APDU size is larger than the maximum packet size allowed on the network, the APDU will be broken up into fragments (NPDUs) for transmission, provided that fragmentation has been enabled by setting the ZigBee network parameter *Maximum Number of Transmitted Simultaneous Fragmented Messages* to a non-zero value.

If data is sent using this function to a destination for which a route has not already been established, the data will not be sent and a route discovery will be performed instead. In this case, the function will return ZPS_NWK_ENUM_ROUTE_ERROR and must later be re-called to send the data (see Note under "Unicast" on page 84).

Once the sent data has reached the first hop node in the route to its destination, a ZPS_EVENT_APS_DATA_CONFIRM event will be generated on the local node. Then, once an acknowledgement has been received from the destination node, a ZPS_EVENT_APS_DATA_ACK will be generated on the sending node.

Security (encyption/decryption) can be applied to the APDU, where this security can be implemented at the Application layer or the network (ZigBee) layer, or both.

Parameters

hAPduInst	Handle of APDU instance to be sent
u16ClusterId	Identifier of relevant output cluster on source endpoint
u8SrcEndpoint	Source endpoint number (1-240) on local node
u8DstEndpoint	Destination endpoint number (1-240) on remote node
u16DstAddr	Network address of destination node

eSecurityMode	Security mode for data transfer: ZPS_E_APL_AF_UNSECURE (no security enabled) ZPS_E_APL_AF_SECURE (Application-level security using link key and network key) ZPS_E_APL_AF_SECURE_NWK (Network-level security using network key) ZPS_E_APL_AF_SECURE|ZPS_E_APL_AF_EXT_NONCE (Application-level security using link key and network key with the extended NONCE included in the frame) ZPS_E_APL_AF_WILD_PROFILE (May be combined with above flags using OR operator. Sends the message using the wild card profile (0xFFFF) instead of the profile in the associated Simple descriptor)
u8Radius	Maximum number of hops permitted to destination node (zero value specifies that default maximum is to be used)
**pu8SeqNum*	Pointer to location to receive sequence number assigned to data transfer request. If not required, set to NULL

Returns

ZPS_E_SUCCESS

APS return codes, listed and described in Section 10.2.2

NWK return codes, listed and described in Section 10.2.3

MAC return codes, listed and described in Section 10.2.4

ZPS_eAplAfUnicastIeeeAckDataReq

```
ZPS_teStatus ZPS_eAplAfUnicastIeeeAckDataReq(
                    PDUM_thAPduInstance hAPduInst,
                    uint16 u16ClusterId,
                    uint8 u8SrcEndpoint,
                    uint8 u8DstEndpoint,
                    uint64 u64DestAddr,
                    ZPS_teAplAfSecurityMode eSecurityMode,
                    uint8 u8Radius,
                    uint8 *pu8SeqNum);
```

Description

This function submits a request to send data to a remote node (unicast), using the remote node's IEEE (MAC) address, and requires an acknowledgement to be returned by the remote node once the data reaches its destination. You must specify the local endpoint and output cluster from which the data originates (the cluster must be in the Simple descriptor for the endpoint), as well as the IEEE address of the remote node and the destination endpoint on the node.

The data is sent in an Application Protocol Data Unit (APDU) instance, which can be allocated using the PDUM function **PDUM_hAPduAllocateAPduInstance()** and then written to using **PDUM_u16APduInstanceWriteNBO()**.

If the APDU size is larger than the maximum packet size allowed on the network, the APDU will be broken up into fragments (NPDUs) for transmission, provided that fragmentation has been enabled by setting the ZigBee network parameter *Maximum Number of Transmitted Simultaneous Fragmented Messages* to a non-zero value.

If data is sent using this function to a destination for which a route has not already been established, the data will not be sent and a route discovery will be performed instead. In this case, the function will return ZPS_NWK_ENUM_ROUTE_ERROR and must later be re-called to send the data (see Note under "Unicast" on page 84).

Once the sent data has reached the first hop node in the route to its destination, a ZPS_EVENT_APS_DATA_CONFIRM event will be generated on the local node. Then, once an acknowledgement has been received from the destination node, a ZPS_EVENT_APS_DATA_ACK will be generated on the sending node.

Security (encyption/decryption) can be applied to the APDU, where this security can be implemented at the Application layer or the network (ZigBee) layer, or both.

Parameters

hAPduInst	Handle of APDU instance to be sent
u16ClusterId	Identifier of relevant output cluster on source endpoint
u8SrcEndpoint	Source endpoint number (1-240) on local node
u8DstEndpoint	Destination endpoint number (1-240) on remote node
u64DestAddr	IEEE (MAC) address of destination node

eSecurityMode	Security mode for data transfer: ZPS_E_APL_AF_UNSECURE (no security enabled) ZPS_E_APL_AF_SECURE (Application-level security using link key and network key) ZPS_E_APL_AF_SECURE_NWK (Network-level security using network key) ZPS_E_APL_AF_SECURE|ZPS_E_APL_AF_EXT_NONCE (Application-level security using link key and network key with the extended NONCE included in the frame) ZPS_E_APL_AF_WILD_PROFILE (May be combined with above flags using OR operator. Sends the message using the wild card profile (0xFFFF) instead of the profile in the associated Simple descriptor)
u8Radius	Maximum number of hops permitted to destination node (zero value specifies that default maximum is to be used)
**pu8SeqNum*	Pointer to location to receive sequence number assigned to data transfer request. If not required, set to NULL

Returns

ZPS_E_SUCCESS

APS return codes, listed and described in Section 10.2.2

NWK return codes, listed and described in Section 10.2.3

MAC return codes, listed and described in Section 10.2.4

ZPS_eAplAfGroupDataReq

```
ZPS_teStatus ZPS_eAplAfGroupDataReq(
                    PDUM_thAPduInstance hAPduInst,
                    uint16 u16ClusterId,
                    uint8 u8SrcEndpoint,
                    uint16 u16DstGroupAddr,
                    ZPS_teAplAfSecurityMode eSecurityMode,
                    uint8 u8Radius,
                    uint8 *pu8SeqNum);
```

Description

This function submits a request to send data to a group of endpoints located on one or more nodes (group multicast). You must specify the local endpoint and output cluster from which the data originates (the cluster must be in the Simple descriptor for the endpoint) as well as the 'group address' of the group of destination endpoints. A group is set up using the function **ZPS_eAplZdoGroupEndpointAdd()**. The data is actually broadcast to all network nodes and each recipient node assesses whether it has endpoints in the specified group.

The data is sent in an Application Protocol Data Unit (APDU) instance, which can be allocated using the PDUM function **PDUM_hAPduAllocateAPduInstance()** and then written to using **PDUM_u16APduInstanceWriteNBO()**.

If the APDU size is larger than the maximum packet size allowed on the network, this function call will fail (and return ZPS_E_ADSU_TOO_LONG).

Once the data has been transmitted, a ZPS_EVENT_APS_DATA_CONFIRM event will be generated on the local node.

Security (encyption/decryption) can be applied to the APDU, where this security can be implemented at the Application layer or the network (ZigBee) layer, or both.

Parameters

hAPduInst	Handle of APDU instance to be sent
u16ClusterId	Identifier of relevant output cluster on source endpoint
u8SrcEndpoint	Source endpoint number (1-240) on local node
u16DstGroupAddr	Group address of destination endpoints
eSecurityMode	Security mode for data transfer, one of: ZPS_E_APL_AF_UNSECURE (no security enabled) ZPS_E_APL_AF_SECURE_NWK (Network-level security using network key) ZPS_E_APL_AF_WILD_PROFILE (May be combined with above flags using OR operator. Sends the message using the wild card profile (0xFFFF) instead of the profile in the associated Simple descriptor)
u8Radius	Maximum number of hops permitted to destination node (zero value specifies that default maximum is to be used)

**pu8SeqNum*	Pointer to location to receive sequence number assigned to data transfer request. If not required, set to NULL

Returns

ZPS_E_SUCCESS

APS return codes, listed and described in Section 10.2.2

NWK return codes, listed and described in Section 10.2.3

MAC return codes, listed and described in Section 10.2.4

ZPS_eAplAfBroadcastDataReq

```
ZPS_teStatus ZPS_eAplAfBroadcastDataReq(
            PDUM_thAPduInstance hAPduInst,
            uint16 u16ClusterId,
            uint8 u8SrcEndpoint,
            uint8 u8DstEndpoint,
            ZPS_teAplAfBroadcastMode eBroadcastMode,
            ZPS_teAplAfSecurityMode eSecurityMode,
            uint8 u8Radius,
            uint8 *pu8SeqNum);
```

Description

This function submits a request to send data to all network nodes that conform to the specified broadcast mode. You must specify the local endpoint and output cluster from which the data originates (the cluster must be in the Simple descriptor for the endpoint), as well as the destination endpoint(s) on the remote nodes.

The data is sent in an Application Protocol Data Unit (APDU) instance, which can be allocated using the PDUM function **PDUM_hAPduAllocateAPduInstance()** and then written to using **PDUM_u16APduInstanceWriteNBO()**.

If the APDU size is larger than the maximum packet size allowed on the network, this function call will fail (and return ZPS_E_ADSU_TOO_LONG).

Following this function call, the APDU may be broadcast up to four times by the source node (in addition, the APDU may be subsequently re-broadcast up to four times by each intermediate routing node). If the transmission is successful, the event ZPS_EVENT_APS_DATA_CONFIRM will be generated on the local node.

Security (encyption/decryption) can be applied to the APDU, where this security can be implemented at the Application layer or the network (ZigBee) layer, or both.

Parameters

hAPduInst	Handle of APDU instance to be sent
u16ClusterId	Identifier of relevant output cluster on source endpoint
u8SrcEndpoint	Source endpoint number (1-240) on local node
u8DstEndpoint	Destination endpoint number (1-240) on remote node, or 255 for all endpoints on node
eBroadcastMode	Type of broadcast, one of: ZPS_E_BROADCAST_ALL (all nodes) ZPS_E_BROADCAST_ALL RX_ON (all nodes with radio receiver permanently enabled) ZPS_E_BROADCAST_ZC_ZR (all Routers and Co-ordinator)

eSecurityMode	Security mode for data transfer: ZPS_E_APL_AF_UNSECURE (no security enabled) ZPS_E_APL_AF_SECURE_NWK (Network-level security using network key) ZPS_E_APL_AF_WILD_PROFILE (May be combined with above flags using OR operator. Sends the message using the wild card profile (0xFFFF) instead of the profile in the associated Simple descriptor)
u8Radius	Maximum number of hops permitted to destination node (zero value specifies that default maximum is to be used)
**pu8SeqNum*	Pointer to location to receive sequence number assigned to data transfer request. If not required, set to NULL

Returns

ZPS_E_SUCCESS

APS return codes, listed and described in Section 10.2.2

NWK return codes, listed and described in Section 10.2.3

MAC return codes, listed and described in Section 10.2.4

ZPS_eAplAfBoundDataReq

```
ZPS_teStatus ZPS_eAplAfBoundDataReq(
                    PDUM_thAPduInstance hAPduInst,
                    uint16 u16ClusterId,
                    uint8 u8SrcEndpoint,
                    ZPS_teAplAfSecurityMode eSecurityMode,
                    uint8 u8Radius,
                    uint8 *pu8SeqNum);
```

Description

This function submits a request to send data to all nodes/endpoints to which the source node/endpoint has been previously bound (using the binding functions, described in Section 8.1.3). You must specify the local endpoint and output cluster from which the data originates (the cluster must be in the Simple descriptor for the endpoint).

The data is sent in an Application Protocol Data Unit (APDU) instance, which can be allocated using the PDUM function **PDUM_hAPduAllocateAPduInstance()** and then written to using **PDUM_u16APduInstanceWriteNBO()**.

If the APDU size is larger than the maximum packet size allowed on the network, this function call will fail (and return ZPS_E_ADSU_TOO_LONG).

Once the sent data has reached the first hop node in the route to its destination(s), a ZPS_EVENT_BIND_REQUEST_SERVER event will be generated on the local node. This event reports the status of the bound transmission, including the number of bound endpoints for which the transmission has failed.

Security (encyption/decryption) can be applied to the APDU, where this security can be implemented at the Application layer or the network (ZigBee) layer, or both.

Parameters

hAPduInst	Handle of APDU instance to be sent
u16ClusterId	Identifier of relevant output cluster on source endpoint
u8SrcEndpoint	Source endpoint number (1-240) on local node
eSecurityMode	Security mode for data transfer: ZPS_E_APL_AF_UNSECURE (no security enabled) ZPS_E_APL_AF_SECURE (Application-level security using link key and network key) ZPS_E_APL_AF_SECURE_NWK (Network-level security using network key) ZPS_E_APL_AF_SECURE | ZPS_E_APL_AF_EXT_NONCE (Application-level security using link key and network key with the extended NONCE included in the frame) ZPS_E_APL_AF_WILD_PROFILE (May be combined with above flags using OR operator. Sends the message using the wild card profile (0xFFFF) instead of the profile in the associated Simple descriptor)

u8Radius	Maximum number of hops permitted to destination node (zero value specifies that default maximum is to be used)
**pu8SeqNum*	Pointer to location to receive sequence number assigned to data transfer request. If not required, set to NULL.

Returns

ZPS_E_SUCCESS

APS return codes, listed and described in Section 10.2.2

NWK return codes, listed and described in Section 10.2.3

MAC return codes, listed and described in Section 10.2.4

ZPS_eAplAfBoundAckDataReq

```
ZPS_teStatus ZPS_eAplAfBoundAckDataReq(
                    PDUM_thAPduInstance hAPduInst,
                    uint16 u16ClusterId,
                    uint8 u8SrcEndpoint,
                    ZPS_teAplAfSecurityMode eSecurityMode,
                    uint8 u8Radius,
                    uint8 *pu8SeqNum);
```

Description

This function submits a request to send data to all nodes/endpoints to which the source node/endpoint has been previously bound (using the binding functions, described in Section 8.1.3) and requires an acknowledgement to be returned by the remote node(s) once the data reaches its destination(s). You must specify the local endpoint and output cluster from which the data originates (the cluster must be in the Simple descriptor for the endpoint).

The data is sent in an Application Protocol Data Unit (APDU) instance, which can be allocated using the PDUM function **PDUM_hAPduAllocateAPduInstance()** and then written to using **PDUM_u16APduInstanceWriteNBO()**.

If the APDU size is larger than the maximum packet size allowed on the network, the APDU will be broken up into fragments (NPDUs) for transmission, provided that fragmentation has been enabled by setting the ZigBee network parameter *Maximum Number of Transmitted Simultaneous Fragmented Messages* to a non-zero value.

Once the sent data has reached its final destination node(s), a ZPS_EVENT_BIND_REQUEST_SERVER event will be generated on the local node. This event reports the status of the bound transmission, including the number of bound endpoints for which the transmission has failed.

Security (encyption/decryption) can be applied to the APDU, where this security can be implemented at the Application layer or the network (ZigBee) layer, or both.

Parameters

hAPduInst	Handle of APDU instance to be sent
u16ClusterId	Identifier of relevant output cluster on source endpoint
u8SrcEndpoint	Source endpoint number (1-240) on local node
eSecurityMode	Security mode for data transfer: ZPS_E_APL_AF_UNSECURE (no security enabled) ZPS_E_APL_AF_SECURE (Application-level security using link key and network key) ZPS_E_APL_AF_SECURE_NWK (Network-level security using network key) ZPS_E_APL_AF_SECURE | ZPS_E_APL_AF_EXT_NONCE (Application-level security using link key and network key with the extended NONCE included in the frame)

	ZPS_E_APL_AF_WILD_PROFILE (May be combined with above flags using OR operator. Sends the message using the wild card profile (0xFFFF) instead of the profile in the associated Simple descriptor)
u8Radius	Maximum number of hops permitted to destination node (zero value specifies that default maximum is to be used)
**pu8SeqNum*	Pointer to location to receive sequence number assigned to data transfer request. If not required, set to NULL.

Returns

ZPS_E_SUCCESS

APS return codes, listed and described in Section 10.2.2

NWK return codes, listed and described in Section 10.2.3

MAC return codes, listed and described in Section 10.2.4

ZPS_eAplAfInterPanDataReq

```
ZPS_teStatus ZPS_eAplAfInterPanDataReq(
                    PDUM_thAPduInstance hAPduInst,
                    uint16 u16ClusterId,
                    uint16 u16ProfileId,
                    ZPS_tsInterPanAddress *psDstAddr,
                    uint8 u8Handle);
```

Description

This function submits a request to send data to one or more nodes in another ZigBee PRO network - that is, to implement an inter-PAN transmission. The destination for the data is specified in a structure (detailed in Section 7.2.3.3) which contains:

- PAN ID of destination network (a broadcast to all reachable ZigBee PRO networks can also be configured)
- Address of destination node (this can be an IEEE/MAC or network address for a single node, a group address for multiple nodes or a broadcast address for all nodes)

The data is sent in an Application Protocol Data Unit (APDU) instance, which can be allocated using the PDUM function **PDUM_hAPduAllocateAPduInstance()** and then written to using **PDUM_u16APduInstanceWriteNBO()**.

If the APDU size is larger than the maximum packet size allowed on the local network, this function call will fail (and return ZPS_E_ADSU_TOO_LONG).

Once the sent data has reached the first hop node in the route to its destination, a ZPS_EVENT_APS_INTERPAN_DATA_CONFIRM event will be generated on the local node (in the case of a broadcast or group multicast, this event is simply generated once the data has been sent from the local node).

Security (encyption/decryption) cannot be applied to inter-PAN transmissions.

Parameters

hAPduInst	Handle of APDU instance to be sent
u16ClusterId	Identifier of cluster for which data is intended at destination (must be a cluster of the application profile specified below)
u16ProfileId	Identifier of application profile for which data is intended at destination
psDstAddr	Pointer to stucture containing destination PAN ID and address (see Section 7.2.3.3)
u8Handle	Handle for internal use (set to any value)

Returns

ZPS_E_SUCCESS

ZPS_APL_APS_E_ILLEGAL_REQUEST

MAC return codes, listed and described in Section 10.2.4

ZPS_u8AplGetMaxPayloadSize

```
uint8 ZPS_u8AplGetMaxPayloadSize(void *pvApl,
                                 uint16 u16Addr);
```

Description

This function obtains the effective payload size, in bytes, within an IEEE802.15.4 data frame to be sent to the node with the specified network address. The handle of the relevant Application layer instance must also be specified, which can be obtained using **ZPS_pvAplZdoGetAplHandle()**.

An IEEE802.15.4 data frame contains 127 bytes, but the effective payload is reduced by the various IEEE802.15.4 and ZigBee headers. The function returns the size of the payload available for data but does not take into account bytes needed for ZCL cluster headers (so may not reflect the exact amount of space available for data).

Parameters

pvApl	Handle of handle for the Application layer instance
u16Addr	16-bit network address of node to which data is to be sent

Returns

Number of data frame payload bytes available for data (ignoring ZCL headers)

7.1.3 Endpoint Functions

The AF Endpoint functions are used to control and monitor the states of endpoints on the local node.

The functions are listed below, along with their page references:

ZPS_vAplAfSetEndpointState

```
ZPS_teStatus ZPS_eAplAfSetEndpointState(
                                        uint8 u8Endpoint,
                                        bool bEnabled);
```

Description

This function puts the specified endpoint on the local node into the specified state (enabled or disabled).

Parameters

u8Endpoint	Endpoint number (on local node)
bEnabled	State in which to put endpoint, one of: TRUE: enable endpoint FALSE: disable endpoint

Returns

ZPS_E_SUCCESS (endpoint state successfully set)

APS return codes, listed and described in Section 10.2.2

NWK return codes, listed and described in Section 10.2.3

MAC return codes, listed and described in Section 10.2.4

ZPS_eAplAfGetEndpointState

```
ZPS_teStatus ZPS_eAplAfGetEndpointState(
                                        uint8 u8Endpoint,
                                        bool *pbEnabled);
```

Description

This function obtains the current state (enabled or disabled) of the specified endpoint on the local node.

Parameters

u8Endpoint	Endpoint number (on local node)
**pbEnabled*	Pointer to location to receive endpoint state. The returned state is one of: TRUE: endpoint enabled FALSE: endpoint disabled

Returns

ZPS_E_SUCCESS

APS return codes, listed and described in Section 10.2.2

NWK return codes, listed and described in Section 10.2.3

MAC return codes, listed and described in Section 10.2.4

ZPS_eAplAfSetEndpointDiscovery

```
ZPS_teStatus ZPS_eAplAfSetEndpointDiscovery(
                                    uint8 u8Endpoint,
                                    uint16 u16ClusterId,
                                    bool bOutput,
                                    bool bDiscoverable);
```

Description

This function sets the discoverable state of the specified cluster of the specified endpoint on the local node - that is, whether the cluster/endpoint will be included in 'device discoveries' initiated on the network.

If the cluster/endpoint is discoverable, it will appear in the Simple descriptor of the local node and will also be included in match results requested using the function **ZPS_eAplZdpMatchDescRequest()**.

The initial discoverable state of the cluster/endpoint is pre-set using the ZPS Configuration Editor (see Chapter 12).

Parameters

u8Endpoint	Endpoint number (on local node)
u16ClusterId	Cluster ID
bOutput	Type of cluster (output or input), one of: TRUE: Output cluster FALSE: Input cluster
bDiscoverable	Discoverable state to set, one of: TRUE: Discoverable FALSE: Not discoverable

Returns

ZPS_E_SUCCESS

APS return codes, listed and described in Section 10.2.2

NWK return codes, listed and described in Section 10.2.3

MAC return codes, listed and described in Section 10.2.4

ZPS_eAplAfGetEndpointDiscovery

```
ZPS_teStatus ZPS_eAplAfGetEndpointDiscovery(
                              uint8 u8Endpoint,
                              uint16 u16ClusterId,
                              bool bOutput,
                              bool_t *pbDiscoverable);
```

Description

This function obtains the discoverable state of the specified cluster of the specified endpoint on the local node - that is, whether the cluster/endpoint will be included in 'device discoveries' initiated on the network.

If the cluster/endpoint is discoverable, it will appear in the Simple descriptor of the local node and will also be included in match results requested using the function **ZPS_eAplZdpMatchDescRequest()**.

The initial discoverable state of the cluster/endpoint is pre-set using the ZPS Configuration Editor (see Chapter 12). The state can subsequently be changed at run-time using the function **ZPS_eAplAfSetEndpointDiscovery()**.

Parameters

u8Endpoint	Endpoint number (on local node)
u16ClusterId	Cluster ID
bOutput	Type of cluster (output or input), one of: TRUE: Output cluster FALSE: Input cluster
**pbDiscoverable*	Pointer to location to receive discoverable state, which will be one of: TRUE: Discoverable FALSE: Not discoverable

Returns

ZPS_E_SUCCESS

APS return codes, listed and described in Section 10.2.2

NWK return codes, listed and described in Section 10.2.3

MAC return codes, listed and described in Section 10.2.4

7.1.4 Descriptor Functions

The AF Descriptor functions allow ZigBee descriptors for the local node to be copied to and from the context area of the ZigBee PRO stack.

The functions are listed below, along with their page references:

ZPS_eAplAfGetNodeDescriptor

```
ZPS_teStatus ZPS_eAplAfGetNodeDescriptor(
                        ZPS_tsAplAfNodeDescriptor *psDesc);
```

Description

This function copies the Node descriptor (for the local node) from the context area of the stack to the specified structure (the descriptor is returned through the function's parameter).

Parameters

******psDesc*	Pointer to structure (see Section 7.2.1.1) to receive Node descriptor

Returns

ZPS_E_SUCCESS

APS return codes, listed and described in Section 10.2.2

NWK return codes, listed and described in Section 10.2.3

MAC return codes, listed and described in Section 10.2.4

ZPS_eAplAfGetNodePowerDescriptor

```
ZPS_teStatus ZPS_eAplAfGetNodePowerDescriptor(
                ZPS_tsAplAfNodePowerDescriptor *psDesc);
```

Description

This function copies the Node Power descriptor (for the local node) from the context area of the stack to the specified structure (the descriptor is returned through the function's parameter).

Parameters

******psDesc*	Pointer to structure (see Section 7.2.1.2) to receive Node Power descriptor

Returns

ZPS_E_SUCCESS

APS return codes, listed and described in Section 10.2.2

NWK return codes, listed and described in Section 10.2.3

MAC return codes, listed and described in Section 10.2.4

ZPS_eAplAfGetSimpleDescriptor

```
ZPS_teStatus ZPS_eAplAfGetSimpleDescriptor(
                    uint8 u8Endpoint,
                    ZPS_tsAplAfSimpleDescriptor *psDesc);
```

Description

This function copies the Simple descriptor for the specified endpoint (on the local node) from the context area of the stack to the specified structure (the descriptor is returned through the function's parameter).

Parameters

psDesc	Pointer to structure (see Section 7.2.1.3) to receive Simple descriptor

Returns

ZPS_E_SUCCESS

APS return codes, listed and described in Section 10.2.2

NWK return codes, listed and described in Section 10.2.3

MAC return codes, listed and described in Section 10.2.4

7.1.5 Other Functions

This section described other functions in the AF API.

The functions are listed below, along with their page references:

ZPS_vSaveAllZpsRecords

```
void ZPS_vSaveAllZpsRecords(void);
```

Description

This function saves to Non-Volatile Memory (NVM) all the NVM records related to the ZigBee PRO stack. This function must be used in conjunction with the Non-Volatile Memory Manager (NVM), which is described in the *JN51xx Core Utilities User Guide (JN-UG-3133)*.

Parameters

None

Returns

None

ZPS_bAplAfSetEndDeviceTimeout

```
bool ZPS_bAplAfSetEndDeviceTimeout
                    (teZedTimeout eZedTimeout);
```

Description

This function can be used on an End Device to configure a timeout period for the End Device Ageing mechanism, which is described in Section 5.10.1.

The End Device will communicate this timeout period to its parent on joining the network. The parent applies this timeout to the 'keep-alive' packets sent from the End Device child using the function **ZPS_eAplAfSendKeepAlive()** - if the parent does not receive a keep-alive packet from the End Device before the timeout expires then the parent assumes the End Device is no longer active and discards it.

Parameters

eZedTimeout	Enumeration indicating timeout period to be set - one of: ZED_TIMEOUT_10_SEC (10 seconds) ZED_TIMEOUT_2_MIN (2 minutes) ZED_TIMEOUT_4_MIN (4 minutes) ZED_TIMEOUT_8_MIN (8 minutes) ZED_TIMEOUT_16_MIN (16 minutes) ZED_TIMEOUT_32_MIN (32 minutes) ZED_TIMEOUT_64_MIN (64 minutes) ZED_TIMEOUT_128_MIN (128 minutes) ZED_TIMEOUT_256_MIN (256 minutes) ZED_TIMEOUT_512_MIN (512 minutes) ZED_TIMEOUT_1024_MIN (1024 minutes) ZED_TIMEOUT_2048_MIN (2048 minutes) ZED_TIMEOUT_4096_MIN (4096 minutes) ZED_TIMEOUT_8192_MIN (8192 minutes) ZED_TIMEOUT_16384_MIN (16384 minutes)

Returns

TRUE - timeout successfully set

FALSE - timeout not set

ZPS_eAplAfSendKeepAlive

```
ZPS_teStatus ZPS_eAplAfSendKeepAlive(void);
```

Description

This function can be used on an End Device to send a ‘keep-alive’ packet to its parent as part of the End Device Ageing mechanism, which is described in Section 5.10.1. This packet informs the parent that the End Device is still active, so that the parent will not discard the child.

The parent must receive at least one keep-alive packet from the End Device within the timeout period defined using the function **ZPS_bAplAfSetEndDeviceTimeout()**. Otherwise, the parent will assume that the child is no longer active and discard the child. It is recommended that at least three keep-alive packets are sent within the timeout period to ensure that the End Device child is not accidently discarded due to missed keep-alive packets.

A keep-alive packet can take the form of a MAC Data Poll or an End Device Timeout Request, as required by the parent - the keep-alive packet type is configured in the NIB on the parent but, by default, both packets types are configured to be acceptable in the NXP software. This function automatically sends the appropriate keep-alive packet type but when both packet types are acceptable, a Data Poll is sent. Both packet types have the effect of re-starting the timeout for the End Device on the parent. When a Data Poll packet is used, the parent may also return pending data to the End Device, indicated by a ZPS_EVENT_AF_DATA_INDICATION event on the End Device.

Parameters

None

Returns

ZPS_E_SUCCESS

APS return codes, listed and described in Section 10.2.2

NWK return codes, listed and described in Section 10.2.3

MAC return codes, listed and described in Section 10.2.4

7.2 AF Structures

This section describes the structures of the Application Framework (AF) API.

These include the following categories of structure:

- Descriptor structures - see Section 7.2.1
- Event structures - see Section 7.2.2
- Other structures - see Section 7.2.3

7.2.1 Descriptor Structures

These structures are used to represent the following descriptors that contain information about the host node:

- Node descriptor
- Node Power descriptor
- Simple descriptor

The structures are listed below, along with their page references.

7.2.1.1 ZPS_tsAplAfNodeDescriptor

The AF Node descriptor structure `ZPS_tsAplAfNodeDescriptor` is shown below.

```
typedef struct {
    uint32                     : 8; /* padding */
    uint32 eLogicalType        : 3;
    uint32 bComplexDescAvail   : 1;
    uint32 bUserDescAvail      : 1;
    uint32 eReserved           : 3; /* reserved */
    uint32 eFrequencyBand      : 5;
    uint32 eApsFlags           : 3;
    uint32 u8MacFlags          : 8;
    uint16 u16ManufacturerCode;
    uint8 u8MaxBufferSize;
    uint16 u16MaxRxSize;
    uint16 u16ServerMask;
    uint16 u16MaxTxSize;
    uint8 u8DescriptorCapability;
} ZPS_tsAplAfNodeDescriptor;
```

where:

- eLogicalType contains 3 bits (bits 0-2) indicating the ZigBee device type of the node, as follows:
  - 000: Co-ordinator
  - 001: Router
  - 010: End Device
- bComplexDescAvail is set to 1 if there is a Complex descriptor available for node.
- bUserDescAvail is set to 1 if there is a User descriptor available for node.
- eReserved is reserved.
- eFrequencyBand contains 5 bits detailing the frequency bands supported by the node, as follows (a bit is set to 1 if the corresponding band is supported):
  - Bit 0: 868-868.6 MHz
  - Bit 2: 902-928 MHz
  - Bit 3: 2400-2483.5 MHz
  - Bits 1 and 4 are reserved
- eApsFlags is not currently supported and set to zero.
- eMacFlags contains 8 bits (bits 0-7) indicating the node capabilities, as required by the IEEE 802.15.4 MAC sub-layer. These node capability flags are described in Table 8 on page 199.
- u16ManufacturerCode contains 16 bits (bits 0-15) indicating the manufacturer code for the node, where this code is allocated to the manufacturer by the ZigBee Alliance.
- u8MaxBufferSize is the maximum size, in bytes, of an NPDU (Network Protocol Data Unit).
- u16MaxRxSize is the maximum size, in bytes, of an APDU (Application Protocol Data Unit). This value can be greater than the value of u8MaxBufferSize, due to the fragmentation of an APDU into NPDUs.
- u16ServerMask contains 8 bits (bits 0-7) indicating the server status of the node. This server mask is detailed in Table 13 on page 360.
- u16MaxTxSize is the maximum size, in bytes, of the ASDU (Application Sub-layer Data Unit) in which a message can be sent (the message may actually be transmitted in smaller fragments)
- u8DescriptorCapability contains 8 bits (bits 0-7) indicating the properties of the node that can be used by other nodes in network discovery, as follows:

Bit	Description
0	Set to 1 if Extended Active Endpoint List is available on the node, 0 otherwise
1	Set to 1 if Extended Simple Descriptor List is available on the node, 0 otherwise
2-7	Reserved

7.2.1.2 ZPS_tsAplAfNodePowerDescriptor

The AF Node Power descriptor structure `ZPS_tsAplAfNodePowerDescriptor` is shown below.

```
typedef struct {
  uint32 eCurrentPowerMode         : 4;
  uint32 eAvailablePowerSources    : 4;
  uint32 eCurrentPowerSource       : 4;
  uint32 eCurrentPowerSourceLevel  : 4;
} ZPS_tsAplAfNodePowerDescriptor;
```

where:

- `eCurrentPowerMode` contains 4 bits (bits 0-3) indicating the power mode currently used by the node, as follows:
  - 0000: Receiver configured according to “Receiver on when idle” MAC flag in the Node Descriptor (see Section 7.2.1.1)
  - 0001: Receiver switched on periodically
  - 0010: Receiver switched on when stimulated, e.g. by pressing a button
  - All other values are reserved
- `eAvailablePowerSources` contains 4 bits (bits 0-3) indicating the available power sources for the node, as follows (a bit is set to 1 if the corresponding power source is available):
  - Bit 0: Permanent mains supply
  - Bit 1: Rechargeable battery
  - Bit 2: Disposable battery
  - Bit 4: Reserved
- `eCurrentPowerSource` contains 4 bits (bits 0-3) indicating the current power source for the node, as detailed for the element above (the bit corresponding to the current power source is set to 1, all other bits are set to 0).
- `eCurrentPowerSourceLevel` contains 4 bits (bit 0-3) indicating the current level of charge of the node’s power source (mainly useful for batteries), as follows:
  - 0000: Critically low
  - 0100: Approximately 33%
  - 1000: Approximately 66%
  - 1100: Approximately 100% (near fully charged)

7.2.1.3 ZPS_tsAplAfSimpleDescriptor

The AF Simple descriptor structure `ZPS_tsAplAfSimpleDescriptor` is shown below.

```
typedef struct {
  uint16 u16ApplicationProfileId;
  uint16 u16DeviceId;
  uint8  u8DeviceVersion;
  uint8  u8Endpoint;
  uint8  u8InClusterCount;
  uint8  u8OutClusterCount;
  uint16 *pu16InClusterList;
  uint16 *pu16OutClusterList;
} ZPS_tsAplAfSimpleDescriptor;
```

where:

- `u16ApplicationProfileId` is the 16-bit identifier of the ZigBee application profile supported by the endpoint. This must be an application profile identifier issued by the ZigBee Alliance (for Lighting & Occupancy devices, it is 0x0104).
- `u16DeviceId` is the 16-bit identifier of the ZigBee device type supported by the endpoint. This must be a device type identifier issued by the ZigBee Alliance.
- `u8DeviceVersion` contains 4 bits (bits 0-3) representing the version of the supported device description (default is 0000, unless set to another value according to the application profile used).
- `u8Endpoint` is the number, in the range 1-240, of the endpoint to which the Simple descriptor corresponds.
- `u8InClusterCount` is an 8-bit count of the number of input clusters, supported on the endpoint, that will appear in the list pointed to by the `pu16InClusterList` element.
- `u8OutClusterCount` is an 8-bit count of the number of output clusters, supported on the endpoint, that will appear in the `pu16OutClusterList` element.
- `*pu16InClusterList` is a pointer to the list of input clusters supported by the endpoint (for use during the service discovery and binding procedures). This is a sequence of 16-bit values, representing the cluster numbers (in the range 1-240), where the number of values is equal to count `u8InClusterCount`. If this count is zero, the pointer can be set to NULL.
- `*pu16OutClusterList` is a pointer to the list of output clusters supported by the endpoint (for use during the service discovery and binding procedures). This is a sequence of 16-bit values, representing the cluster numbers (in the range 1-240), where the number of values is equal to count `u8OutClusterCount`. If this count is zero, the pointer can be set to NULL.

7.2.2 Event Structures

These stuctures are used to contain events. Event details (type and associated data) are passed to the application in the structure `ZPS_tsAfEvent`. Data structures for the individual event types are contained in the union `ZPS_tuAfEventData`.

Note: Enumerations for the event types are provided in the stucture `ZPS_teAfEventType`. This structure and the associated events are detailed in Section 10.1.

The structures are listed below, along with their page references.

7.2.2.1 ZPS_tsAfEvent

This stucture contains the details of an event.

The `ZPS_tsAfEvent` structure is detailed below.

```
typedef struct {
    ZPS_teAfEventType eType;
    ZPS_tuAfEventData uEvent;
} ZPS_tsAfEvent;
```

where

- `eType` indicates the event type, using the enumerations listed and described in Section 10.1
- `uEvent` is a structure containing the event data from the union of structures detailed in Section 7.2.2.2

7.2.2.2 ZPS_tuAfEventData

This structure is a union of the data structures for the individual events described in Section 7.2.2.3 through to Section 7.2.2.25.

The `ZPS_tuAfEventData` structure is detailed below.

```
typedef union
{
    ZPS_tsAfDataIndEvent                 sApsDataIndEvent;
    ZPS_tsAfDataConfEvent                sApsDataConfirmEvent;
    ZPS_tsAfDataAckEvent                 sApsDataAckEvent;
    ZPS_tsAfNwkFormationEvent            sNwkFormationEvent;
    ZPS_tsAfNwkJoinedEvent               sNwkJoinedEvent;
    ZPS_tsAfNwkJoinFailedEvent           sNwkJoinFailedEvent;
    ZPS_tsAfNwkDiscoveryEvent            sNwkDiscoveryEvent;
    ZPS_tsAfNwkJoinIndEvent              sNwkJoinIndicationEvent;
    ZPS_tsAfNwkLeaveIndEvent             sNwkLeaveIndicationEvent;
    ZPS_tsAfNwkLeaveConfEvent            sNwkLeaveConfirmEvent;
    ZPS_tsAfNwkStatusIndEvent            sNwkStatusIndicationEvent;
    ZPS_tsAfNwkRouteDiscoveryConfEvent   sNwkRouteDiscoveryConfirmEvent;
    ZPS_tsAfPollConfEvent                sNwkPollConfirmEvent;
    ZPS_tsAfNwkEdScanConfEvent           sNwkEdScanConfirmEvent;
    ZPS_tsAfErrorEvent                   sAfErrorEvent;
    ZPS_tsAfZdoBindEvent                 sZdoBindEvent;
    ZPS_tsAfZdoUnbindEvent               sZdoUnbindEvent;
    ZPS_tsAfZdoLinkKeyEvent              sZdoLinkKeyEvent;
    ZPS_tsAfBindRequestServerEvent       sBindRequestServerEvent;
    ZPS_tsAfInterPanDataIndEvent         sApsInterPanDataIndEvent;
    ZPS_tsAfInterPanDataConfEvent        sApsInterPanDataConfirmEvent;
    ZPS_tsAfZdpEvent                     sApsZdpEvent;
} ZPS_tuAfEventData;
```

7.2.2.3 ZPS_tsAfDataIndEvent

This structure is used in the ZPS_EVENT_APS_DATA_INDICATION event, which indicates the arrival of data on the local node.

The `ZPS_tsAfDataIndEvent` structure is detailed below.

```
typedef struct
{
    uint8               u8DstAddrMode;
    ZPS_tuAddress       uDstAddress;
    uint8               u8DstEndpoint;
    uint8               u8SrcAddrMode;
    ZPS_tuAddress       uSrcAddress;
    uint8               u8SrcEndpoint;
    uint16              u16ProfileId;
    uint16              u16ClusterId;
    PDUM_thAPduInstance hAPduInst;
    uint8               eStatus;
    uint8               eSecurityStatus;
    uint8               u8LinkQuality;
    uint32              u32RxTime;
} ZPS_tsAfDataIndEvent;
```

where:

- `u8DstAddrMode` indicates the type of destination address specified through the element `uDstAddress` (see Table 7 below)
- `uDstAddress` is the address of the destination node for the data packet (the type of address is specified using the element `u8DstAddrMode` above)
- `u8DstEndpoint` is the number of the destination endpoint (in range 0-240)
- `u8SrcAddrMode` indicates the type of source address specified through the element `uSrcAddress` (below) - this can be a 64-bit MAC/IEEE address or a 16-bit network address
- `uSrcAddress` is the address of the source node for the data packet (the type of address is specified using the element `u8SrcAddrMode` above)
- `u8SrcEndpoint` is the number of the source endpoint (in range 1-240)
- `u16ProfileId` is the identifier of the ZigBee device profile of the device which can interpret the data
- `u16ClusterId` is the identifier of the cluster (which belongs to the device profile specified in `u16ProfileId`) which is capable of interpreting the data
- `hAPduInst` is the handle of the APDU which contains the data
- `eStatus` is one of the status codes from the NWK layer or MAC layer, detailed in Section 10.2.3 and Section 10.2.4
- `eSecurityStatus` indicates the type of security with which the packet was sent - unsecured (0xAF), secured with network key (0xAC) or secured with link key (0xAB)

- `u8LinkQuality` is a measure of the signal strength of the radio link over which the data packet was sent (for the last hop)
- `u32RxTime` is reserved for future use.

u8DstAddrMode	Code	Description
0x00	ZPS_E_ADDR_MODE_BOUND	Bound endpoint
0x01	ZPS_E_ADDR_MODE_GROUP	16-bit Group address
0x02	ZPS_E_ADDR_MODE_SHORT	16-bit Network (Short) address
0x03	ZPS_E_ADDR_MODE_IEEE	64-bit IEEE/MAC address

Table 7: Addressing Modes

7.2.2.4 ZPS_tsAfDataConfEvent

This structure is used in the ZPS_EVENT_APS_DATA_CONFIRM event, which confirms that a data packet sent by the local node has been successfully passed down the stack to the MAC layer and has made its first hop towards its destination (an acknowledgment has been received from the next hop node).

The `ZPS_tsAfDataConfEvent` structure is detailed below.

```
typedef struct {
    uint8         u8Status;
    uint8         u8SrcEndpoint;
    uint8         u8DstEndpoint;
    uint8         u8DstAddrMode;
    ZPS_tuAddress uDstAddr;
    uint8         u8SequenceNum;
} ZPS_tsAfDataConfEvent;
```

where:

- `u8Status` is one of the status codes from the lower stack layers, detailed in Section 10.2.
- `u8SrcEndpoint` is the number of the (local) source endpoint for the data transfer (in range 1-240)
- `u8DstEndpoint` is the number of the destination endpoint for the data transfer (in range 1-240)
- `u8DstAddrMode` indicates the type of destination address specified through the element `uDstAddr` (see Table 7 on page 194) - only values 0x02 (group address) and 0x03 (network address) are valid in this structure
- `uDstAddr` is the address of the destination node for the data packet (the type of address is specified using the element `u8DstAddrMode` above)
- `u8SequenceNum` is the sequence number of the request that initiated the data transfer

7.2.2.5 ZPS_tsAfDataAckEvent

This structure is used in the ZPS_EVENT_APS_DATA_ACK event, which is generated when an end-to-end acknowledgement is received from the destination node during a data transfer in which an acknowledgement was requested.

```
typedef struct {
    uint8   u8Status;
    uint8   u8SrcEndpoint;
    uint8   u8DstEndpoint;
    uint8   u8DstAddrMode;
    uint16  u16DstAddr;
    uint8   u8SequenceNum;
    uint16  u16ProfileId;
    uint16  u16ClusterId;
} ZPS_tsAfDataAckEvent;
```

where:

- `u8Status` is one of the status codes from the lower stack layers, detailed in Section 10.2
- `u8SrcEndpoint` is the number of the (local) source endpoint for the data transfer (in range 1-240)
- `u8DstEndpoint` is the number of the destination endpoint for the data transfer (in range 1-240)
- `u8DstAddrMode` indicates the type of destination address specified through the element `u16DstAddr` (see Table 7 on page 194) - only values 0x01 (group address) and 0x02 (network address) are valid in this structure
- `u16DstAddr` is the 16-bit address of the destination node for the data transfer and therefore of the node that sent the acknowledgement (the type of address is specified using the element `u8DstAddrMode` above)
- `u8SequenceNum` is the sequence number of the request that initiated the data transfer
- `u16ProfileId` is the identifier of the ZigBee device profile of the device for which the data transfer was intended
- `u16ClusterId` is the identifier of the cluster (which belongs to the device profile specified in `u16ProfileId`) for which the data transfer was intended

7.2.2.6 ZPS_tsAfNwkFormationEvent

This structure is used in the event ZPS_EVENT_NWK_STARTED, which indicates whether the network has been started (on the Co-ordinator).

The `ZPS_tsAfNwkFormationEvent` structure is detailed below.

```
typedef struct {
    uint8 u8Status;
} ZPS_tsAfNwkFormationEvent;
```

where is one of the status codes from the lower stack layers, detailed in Section 10.2.

7.2.2.7 ZPS_tsAfNwkJoinedEvent

This structure is used in the events ZPS_EVENT_NWK_JOINED_AS_ROUTER and ZPS_EVENT_NWK_JOINED_AS_ENDDEVICE, which confirm that the local device (Router or End Device) has successfully joined a network.

The `ZPS_tsAfNwkJoinedEvent` structure reports the network address that the parent has assigned to the new node and is detailed below.

```
typedef struct
{
    uint16 u16Addr;
    bool_t bRejoin;
} ZPS_tsAfNwkJoinedEvent;
```

where:

- `u16Addr` is the 16-bit network address allocated to the joining node.
- `bRejoin` indicates whether the join was a rejoin (TRUE) or a new association (FALSE).

7.2.2.8 ZPS_tsAfNwkJoinFailedEvent

This structure is used in the event ZPS_EVENT_NWK_FAILED_TO_JOIN, which indicates that the local device has failed to join a network.

The `ZPS_tsAfNwkJoinFailedEvent` structure is detailed below.

```
typedef struct
{
    uint8 u8Status;
    bool_t bRejoin;
} ZPS_tsAfNwkJoinFailedEvent;
```

where:

- `u8Status` is one of the status codes from the lower stack layers, detailed in Section 10.2.
- `bRejoin` indicates whether the join attempt was a rejoin (TRUE) or a new association (FALSE).

7.2.2.9 ZPS_tsAfNwkDiscoveryEvent

This structure is used in the ZPS_EVENT_NWK_DISCOVERY_COMPLETE event, which reports the details of the networks detected in a network discovery initiated by a Router or End Device that needs to join a network.

The `ZPS_tsAfNwkDiscoveryEvent` structure is detailed below.

```
typedef struct
{
    uint32 u32UnscannedChannels;
    uint8 eStatus;
    uint8 u8NetworkCount;
    uint8 u8SelectedNetwork;
    ZPS_tsNwkNetworkDescr *psNwkDescriptors;
} ZPS_tsAfNwkDiscoveryEvent;
```

where:

- `u32UnscannedChannels` is a 32-bit bitmap representing the set of channels from the network discovery that had not yet been scanned when this event was generated. Bits 11 to 26 represent the 2400-MHz channels 11 to 26, where 1 indicates channel scanned and 0 indicates channel not yet scanned.
- `estatus` is the status of the network discovery process, returned by the lower layers (see Section 10.2) - MAC_ENUM_SUCCESS, if the discovery was successfully completed.
- `u8NetworkCount` is the number of networks that had been discovered when this event was generated.
- `u8SelectedNetwork` is the index of the recommended network in the array of reported networks (see below).
- `psNwkDescriptors` is a pointer to the network discovery table in the network NIB. The network discovery table contains array of data structures, where each structure contains details of a discovered network. Each array element is a structure of the type `ZPS_tsNwkNetworkDescr`, described in Section 7.2.3.1. The number of array elements is given by `u8NetworkCount`, described above.

7.2.2.10 ZPS_tsAfNwkJoinIndEvent

This structure is used in the event ZPS_EVENT_NWK_NEW_NODE_HAS_JOINED, which notifies a Router or the Co-ordinator that a new child node has joined the network.

The `ZPS_tsAfNwkJoinIndEvent` structure contains information about the new node and is detailed below.

```
typedef struct
{
    uint64 u64ExtAddr;
    uint16 u16NwkAddr;
    uint8  u8Capability;
    uint8  u8Rejoin;
    uint8  u8SecureRejoin;
} ZPS_tsAfNwkJoinIndEvent;
```

where:

- `u64ExtAddr` is the 64-bit IEEE (MAC) address of the joining node
- `u16NwkAddr` is the 16-bit network address assigned to the joining node
- `u8Capability` is a bitmap indicating the operational capabilities of the joining node. This bitmap is detailed in Table 8 below
- `u8Rejoin` indicates the method used to join the network:
    - 0x00 if joined through association
    - 0x01 if joined directly or used orphaning
    - 0x02 if was network rejoin
- `u8SecureRejoin` indicates whether the join was performed in a secure manner - zero represents FALSE and a non-zero value represents TRUE

Bits	Description
0	Co-ordinator capability: 1: Node able to act as Co-ordinator 0: Node not able to act as Co-ordinator
1	Device type: 1: Full-Function Device (FFD) 0: Reduced-Function Device (RFD) An FFD can act as any node type while an RFD cannot act as the network Co-ordinator.
2	Power source: 1: Node is mains-powered 0: Node is not mains-powered
3	Receiver on when idle: 1: Receiver enabled during idle periods 0: Receiver disabled during idle periods to conserve power
4-5	Reserved
6	Security capability: 1: High security 0: Standard security
7	Allocate address: 1: Network address should be allocated to node 0: Network address need not be allocated to node

Table 8: Node Capabilities Bitmap

7.2.2.11 ZPS_tsAfNwkLeaveIndEvent

This structure is used in the ZPS_EVENT_LEAVE_INDICATION event, which indicates that a neighbouring node has left the network or a remote node has requested the local node to leave.

The `ZPS_tsAfNwkLeaveIndEvent` structure is detailed below.

```
typedef struct {
    uint64 u64ExtAddr;
    uint8  u8Rejoin;
} ZPS_tsAfNwkLeaveIndEvent;
```

where:

- `u64ExtAddr` is the 64-bit IEEE (MAC) address of the node that has left the network, or is zero if the local node has been requested to leave the network
- `u8Rejoin` indicates whether the leaving node was requested to attempt a subsequent rejoin of the network - zero represents FALSE and a non-zero value represents TRUE

7.2.2.12 ZPS_tsAfNwkLeaveConfEvent

This structure is used in the event ZPS_EVENT_NWK_LEAVE_CONFIRM, which reports the results of a node leave request issued by the local node.

The `ZPS_tsAfNwkLeaveConfEvent` structure is detailed below.

```
typedef struct {
    uint64 u64ExtAddr;
    uint8  eStatus;
} ZPS_tsAfNwkLeaveConfEvent;
```

where:

- `u64ExtAddr` is the 64-bit IEEE (MAC) address of the leaving node. This value is zero if the local node itself is leaving
- `eStatus` is the leave status returned by the lower layers - ZPS_NWK_ENUM_SUCCESS, if the leave request has been successful

7.2.2.13 ZPS_tsAfNwkStatusIndEvent

This structure is used in the ZPS_EVENT_NWK_STATUS_INDICATION event, which reports status information from the NWK layer of the stack.

The `ZPS_tsAfNwkStatusIndEvent` structure is detailed below.

```
typedef struct {
    uint16  u16NwkAddr;
    uint8   u8Status;
} ZPS_tsAfNwkStatusIndEvent;
```

where:

- `u16NwkAddr` is the 16-bit network address of the node associated with the event
- `u8Status` is one of the status codes from the lower stack layers, detailed in Section 10.2.

7.2.2.14 ZPS_tsAfNwkRouteDiscoveryConfEvent

This structure is used in the ZPS_EVENT_NWK_ROUTE_DISCOVERY_CONFIRM event, which confirms that a route discovery has been performed.

The `ZPS_tsAfNwkRouteDiscoveryConfEvent` structure is detailed below.

```
typedef struct {
    uint16  u16DstAddress;
    uint8   u8Status;
    uint8   u8NwkStatus;
} ZPS_tsAfNwkRouteDiscoveryConfEvent;
```

where:

- `u16DstAddress` is the destination address for which the route discovery confirm event was generated
- `u8Status` is one of the status codes from the MAC layer, detailed in Section 10.2.4
- `u8NwkStatus` is one of the status codes from the NWK layer, detailed in Section 10.2.3

7.2.2.15 ZPS_tsAfPollConfEvent

This structure is used in the ZPS_EVENT_NWK_POLL_CONFIRM event, which reports the completion of a poll request sent from the (local) End Device to its parent.

The `ZPS_tsAfPollConfEvent` structure is detailed below.

```
typedef struct {
    uint8  u8Status;
} ZPS_tsAfPollConfEvent;
```

where `u8Status` is one of the status codes from the lower stack layers, detailed in Section 10.2.

7.2.2.16 ZPS_tsAfNwkEdScanConfEvent

This structure is used in the ZPS_EVENT_NWK_ED_SCAN event, which indicates that an ‘energy detect’ scan in the 2.4-GHz radio band has completed.

The `ZPS_tsAfNwkEdScanConfEvent` structure is defined as:

```
typedef ZPS_tsNwkNlmeCfmEdScan ZPS_tsAfNwkEdScanConfEvent;
```

where `ZPS_tsNwkNlmeCfmEdScan` is described in Section 7.2.3.2.

7.2.2.17 ZPS_tsAfErrorEvent

This structure is used in the ZPS_EVENT_ERROR event, which reports error situations concerning the storage of received messages in APDU instances.

The `ZPS_tsAfErrorEvent` structure is detailed below.

```
typedef struct {
     enum {
         ZPS_ERROR_APDU_TOO_SMALL,
         ZPS_ERROR_APDU_INSTANCES_EXHAUSTED,
         ZPS_ERROR_NO_APDU_CONFIGURED,
         ZPS_ERROR_OS_MESSAGE_QUEUE_OVERRUN
     } eError;

     union {
         struct {
             uint16 u16ProfileId;
             uint16 u16ClusterId;
             uint16 u16SrcAddr;
             uint16 u16DataSize;
             PDUM_thAPdu hAPdu;
             uint8  u8SrcEndpoint;
             uint8  u8DstEndpoint;
         }sAfErrorApdu;

         struct {
             OS_thMessage hMessage;
         } sAfErrorOsMessageOverrun;

     } uErrorData;
} ZPS_tsAfErrorEvent;
```

The member enumerations and structures of the above structure are detailed below.

eError Enumerations

The error enumerations which are part of the ZPS_tsAfErrorEvent structure are listed and described below.

eError Enumeration	Description
ZPS_ERROR_APDU_TOO_SMALL	Allocated APDU instance is too small to accommodate received message. This error is detailed in the structure sAfErrorApdu, which is described below.
ZPS_ERROR_APDU_INSTANCES_EXHAUSTED	The are no APDU instances available to accommodate the received message. This error is detailed in the structure sAfErrorApdu, which is described below.
ZPS_ERROR_NO_APDU_CONFIGURED	No APDU has been configured to accommodate the received message. This error is detailed in the structure sAfErrorApdu, which is described below.
ZPS_ERROR_OS_MESSAGE_QUEUE_OVERRUN	A message queue is full and can accept no more messages. This error is detailed in the structure sAfErrorOsMessageOverrun, which is described below.

Table 9: eError Enumerations

sAfErrorApdu

This structure is used in the following errors:

- ZPS_ERROR_APDU_TOO_SMALL, which reports that the allocated APDU instance is too small to store a received message
- ZPS_ERROR_APDU_INSTANCES_EXHAUSTED, which reports that there are no allocated APDU instances left to store a received message
- ZPS_ERROR_NO_APDU_CONFIGURED, which reports that no APDU has been configured to store the received message

The sAfErrorApdu structure is detailed below.

```
struct {
    uint16 u16ProfileId;
    uint16 u16ClusterId;
    uint16 u16SrcAddr;
    uint16 u16DataSize;
    PDUM_thAPdu hAPdu;
    uint8  u8SrcEndpoint;
    uint8  u8DstEndpoint;
}sAfErrorApdu;
```

where:

- u16ProfileId is the identifier of the ZigBee application profile associated with the source and destination endpoints for the message
- u16ClusterId is the identifier of the cluster associated with the source and destination endpoints for the message
- u16SrcAddr is the 16-bit network address of the source node of the message

- `u16DataSize` is the size of the received message, in bytes
- `hAPdu` is the handle of the local APDU pool from which the APDU instance comes
- `u8SrcEndpoint` is the number of the source endpoint of the message
- `u8DstEndpoint` is the number of the destination endpoint of the message

sAfErrorOsMessageOverrun

This structure is used in the ZPS_ERROR_OS_MESSAGE_QUEUE_OVERRUN error, which indicates that a message queue is full and can accept no more messages.

The `sAfErrorOsMessageOverrun` structure is detailed below.

```
struct {
   OS_thMessage hMessage;
} sAfErrorOsMessageOverrun;
```

where `hMessage` is the handle of the message type for the queue which is full.

7.2.2.18 ZPS_tsAfZdoBindEvent

This structure is used in the ZPS_EVENT_ZDO_BIND event, which indicates that the local node has been successfully bound to one or more remote nodes.

The `ZPS_tsAfZdoBindEvent` structure is detailed below.

```
typedef struct {
    ZPS_tuAddress uDstAddr;
    uint8 u8DstAddrMode;
    uint8 u8SrcEp;
    uint8 u8DstEp;
} ZPS_tsAfZdoBindEvent;
```

where

- `uDstAddr` is the address of the remote node for the binding (the type of address is specified using the element `u8DstAddrMode` above)
- `u8DstAddrMode` indicates the type of address specified through the element `uDstAddr` (see Table 7 on page 194)
- `u8SrcEp` is the number of the source endpoint for the binding (in range 1-240)
- `u8DstEp` is the number of the destination endpoint for the binding (in range 1-240)

7.2.2.19 ZPS_tsAfZdoUnbindEvent

This structure is used in the ZPS_EVENT_ZDO_UNBIND event, which indicates that the local node has been successfully unbound from one or more remote nodes.

The `ZPS_tsAfZdoUnbindEvent` structure is defined as:

```
typedef ZPS_tsAfZdoBindEvent ZPS_tsAfZdoUnbindEvent;
```

where `ZPS_tsAfZdoBindEvent` is described in Section 7.2.2.18 (but for this event, the data in the structure relates to unbinding rather than binding).

7.2.2.20 ZPS_tsAfZdoLinkKeyEvent

This structure is used in the ZPS_EVENT_ZDO_LINK_KEY event, which indicates that a new application link key has been received and installed, and is ready for use.

The `ZPS_tsAfZdoLinkKeyEvent` structure is defined as:

```
typedef struct {
    uint64 u64IeeeLinkAddr;
} ZPS_tsAfZdoLinkKeyEvent;
```

where `u64IeeeLinkAddr` is the IEEE/MAC address of the remote device with which the installed link key is valid.

7.2.2.21 ZPS_tsAfBindRequestServerEvent

This structure is used in the ZPS_EVENT_BIND_REQUEST_SERVER event, which reports the status of a data transmission sent from the (local) node to a set of bound endpoints.

The `ZPS_tsAfBindRequestServerEvent` structure is detailed below.

```
typedef struct {
    uint8  u8Status;
    uint8  u8SrcEndpoint;
    uint32 u32FailureCount;
} ZPS_tsAfBindRequestServerEvent;
```

where:

- `u8Status` is the overall status of the bound data transmission:
  - Success (0) indicates that the data packet was successfully transmitted to all bound endpoints
  - Failure (non-zero value) indicates that the data packet was not successfully sent to at least one bound endpoint (see `u32FailureCount` below)
- `u8SrcEndpoint` is the number of the local endpoint from which the data packet was sent
- `u32FailureCount` is the number of bound endpoints for which the transmission failed

7.2.2.22 ZPS_tsAfInterPanDataIndEvent

This structure is used in the ZPS_EVENT_APS_INTERPAN_DATA_INDICATION event, which indicates that an inter-PAN data packet has arrived.

The `ZPS_tsAfInterPanDataIndEvent` structure is detailed below.

```
typedef struct
{
    ZPS_tsInterPanAddress sDstAddr;
    uint8  u8SrcAddrMode;
    uint16 u16SrcPan;
    uint64 u64SrcAddress;
    uint16 u16ProfileId;
    uint16 u16ClusterId;
    PDUM_thAPduInstance hAPduInst;
    uint8 eStatus;
    uint8 u8DstEndpoint;
    uint8 u8LinkQuality;
} ZPS_tsAfInterPanDataIndEvent;
```

where

- `sDstAddr` is a structure of the type `ZPS_tsInterPanAddress` (see Section 7.2.3.3) which contains the PAN ID and address for the destination node(s) of the inter-PAN data packet
- `u8SrcAddrMode` indicates the type of address specified through the element `u64SrcAddress` (see Table 7 on page 194)
- `u16SrcPan` is the PAN ID of the network from which the data packet originates
- `u64SrcAddress` is the address of the node which sent the data packet (the type of address is specified using the element `u8SrcAddrMode` above)
- `u16ProfileId` is the identifier of the application profile for which the data packet is intended
- `u16ClusterId` is the identifier of the cluster for which the data packet is intended
- `hAPduInst` is the handle of the APDU instance for the data packet
- `eStatus` is one of the status codes from the lower stack layers, detailed in Section 10.2
- `u8DstEndpoint` is the number of the destination endpoint for the data packet (in range 1-240)
- `u8LinkQuality` is an LQI value indicating the perceived strength of the radio signal which carried the received data packet

7.2.2.23 ZPS_tsAfInterPanDataConfEvent

This structure is used in the ZPS_EVENT_APS_INTERPAN_DATA_CONFIRM event, which indicates that an inter-PAN communication has been sent by the local node and an acknowledgement has been received from the first hop node (this acknowledgement is not generated in the case of a broadcast).

The `ZPS_tsAfInterPanDataConfEvent` structure is detailed below.

```
typedef struct {
    uint8 u8Status;
    uint8 u8Handle;
} ZPS_tsAfInterPanDataConfEvent;
```

where

- `u8Status` is one of the status codes from the lower stack layers, detailed in Section 10.2.
- `u8Handle` is a handle for internal use

7.2.2.24 ZPS_tsAfTCstatusEvent

This structure is used in the ZPS_EVENT_TC_STATUS event, which indicates whether negotiations to establish a link key with the Trust Centre have been successful and, if so, which key is the active key.

The `ZPS_tsAfTCstatusEvent` structure is detailed below.

```
typedef struct
{
    ZPS_tuTcStatusData  uTcData;
    uint8               u8Status;
}ZPS_tsAfTCstatusEvent;
```

where:

- `uTcData` is dependent on `u8Status` (below) and is either a pointer to the link key descriptor in the case of success or the address of the Trust Centre node in the case of failure. `ZPS_tuTcStatusData` is a union, detailed below.
- `u8Status` indicates the results of the link key negotiations - one of:
    - ZPS_E_SUCCESS (link key successfully established)
    - ZPS_APL_APS_E_SECURITY_FAIL (link key not established)

The `ZPS_tuTcStatusData` structure is detailed below.

```
typedef union {
    ZPS_tsAplApsKeyDescriptorEntry  *pKeyDesc;
    uint64                           u64ExtendedAddress;
} ZPS_tuTcStatusData;
```

where:

- `pKeyDesc` is a pointer to the active link key, if successfully established, which is contained in the structure described in Section 7.2.3.6.
- `u64ExtendedAddress` is the IEEE/MAC address of the Trust Centre node with which link key negotiations failed.

7.2.2.25 ZPS_tsAfZdpEvent

This structure is used when a ZPS_EVENT_APS_DATA_INDICATION event is generated containing a response which is destined for the ZDO at endpoint 0. The application can extract the response data from the event using the function **ZPS_bAplZdpUnpackResponse()** and this structure is used to receive the extracted data.

The `ZPS_tsAfZdpEvent` structure is detailed below.

```
typedef struct {
    uint8 u8SequNumber;
    uint16 u16ClusterId;
    union {
        ZPS_tsAplZdpDeviceAnnceReq sDeviceAnnce;
        ZPS_tsAplZdpMgmtNwkUpdateReq sMgmtNwkUpdateReq;
        ZPS_tsAplZdpMgmtPermitJoiningReq sPermitJoiningReq;
        ZPS_tsAplZdpDiscoveryCacheRsp sDiscoveryCacheRsp;
        ZPS_tsAplZdpDiscoveryStoreRsp sDiscoveryStoreRsp;
        ZPS_tsAplZdpNodeDescStoreRsp sNodeDescStoreRsp;
        ZPS_tsAplZdpActiveEpStoreRsp sActiveEpStoreRsp;
        ZPS_tsAplZdpSimpleDescStoreRsp sSimpleDescStoreRsp;
        ZPS_tsAplZdpRemoveNodeCacheRsp sRemoveNodeCacheRsp;
        ZPS_tsAplZdpEndDeviceBindRsp sEndDeviceBindRsp;
        ZPS_tsAplZdpBindRsp sBindRsp;
        ZPS_tsAplZdpUnbindRsp sUnbindRsp;
        ZPS_tsAplZdpReplaceDeviceRsp sReplaceDeviceRsp;
        ZPS_tsAplZdpStoreBkupBindEntryRsp sStoreBkupBindEntryRsp;
        ZPS_tsAplZdpRemoveBkupBindEntryRsp sRemoveBkupBindEntryRsp;
        ZPS_tsAplZdpBackupSourceBindRsp sBackupSourceBindRsp;
        ZPS_tsAplZdpMgmtLeaveRsp sMgmtLeaveRsp;
        ZPS_tsAplZdpMgmtDirectJoinRsp sMgmtDirectJoinRsp;
        ZPS_tsAplZdpMgmtPermitJoiningRsp sPermitJoiningRsp;
        ZPS_tsAplZdpNodeDescRsp sNodeDescRsp;
        ZPS_tsAplZdpPowerDescRsp sPowerDescRsp;
        ZPS_tsAplZdpSimpleDescRsp sSimpleDescRsp;
        ZPS_tsAplZdpNwkAddrRsp sNwkAddrRsp;
        ZPS_tsAplZdpIeeeAddrRsp sIeeeAddrRsp;
        ZPS_tsAplZdpUserDescConf sUserDescConf;
        ZPS_tsAplZdpSystemServerDiscoveryRsp sSystemServerDiscoveryRsp;
        ZPS_tsAplZdpPowerDescStoreRsp sPowerDescStoreRsp;
        ZPS_tsAplZdpUserDescRsp sUserDescRsp;
        ZPS_tsAplZdpActiveEpRsp sActiveEpRsp;
        ZPS_tsAplZdpMatchDescRsp sMatchDescRsp;
```

```
        ZPS_tsAplZdpComplexDescRsp sComplexDescRsp;
        ZPS_tsAplZdpFindNodeCacheRsp sFindNodeCacheRsp;
        ZPS_tsAplZdpExtendedSimpleDescRsp sExtendedSimpleDescRsp;
        ZPS_tsAplZdpExtendedActiveEpRsp sExtendedActiveEpRsp;
        ZPS_tsAplZdpBindRegisterRsp sBindRegisterRsp;
        ZPS_tsAplZdpBackupBindTableRsp sBackupBindTableRsp;
        ZPS_tsAplZdpRecoverBindTableRsp sRecoverBindTableRsp;
        ZPS_tsAplZdpRecoverSourceBindRsp sRecoverSourceBindRsp;
        ZPS_tsAplZdpMgmtNwkDiscRsp sMgmtNwkDiscRsp;
        ZPS_tsAplZdpMgmtLqiRsp sMgmtLqiRsp;
        ZPS_tsAplZdpMgmtRtgRsp sRtgRsp;
        ZPS_tsAplZdpMgmtBindRsp sMgmtBindRsp;
        ZPS_tsAplZdpMgmtCacheRsp sMgmtCacheRsp;
        ZPS_tsAplZdpMgmtNwkUpdateNotify sMgmtNwkUpdateNotify;
    }uZdpData;
    union {
        ZPS_tsAplZdpBindingTableEntry asBindingTable[5];
        ZPS_tsAplZdpNetworkDescr asNwkDescTable[5];
        ZPS_tsAplZdpNtListEntry asNtList[2];
        ZPS_tsAplDiscoveryCache aDiscCache[5];
        uint16 au16Data[34];
        uint8 au8Data[77];
        uint64 au64Data[9];
    }uLists;
}ZPS_tsAfZdpEvent;
```

where:

- `u8SequNumber` is the sequence number of the ZDP request/response
- `u16ClusterId` is the ID of the cluster to which the request/response relates
- `uZdpData` is a union of the different ZDP request/response types:
    - `sDeviceAnnce` is a structure of the type `ZPS_tsAplZdpDeviceAnnceReq`, described in Section 8.2.2.3
    - `sMgmtNwkUpdateReq` is a structure of the type `ZPS_tsAplZdpMgmtNwkUpdateReq`, described in Section 8.2.2.41
    - `sPermitJoiningReq` is a structure of the type `ZPS_tsAplZdpMgmtPermitJoiningReq`, described in Section 8.2.3.39
    - `sDiscoveryCacheRsp` is a structure of the type `ZPS_tsAplZdpDiscoveryCacheRsp`, described in Section 8.2.3.14
    - `sDiscoveryStoreRsp` is a structure of the type `ZPS_tsAplZdpDiscoveryStoreRsp`, described in Section 8.2.3.15
    - `sNodeDescStoreRsp` is a structure of the type `ZPS_tsAplZdpNodeDescStoreRsp`, described in Section 8.2.3.16
    - `sActiveEpStoreRsp` is a structure of the type `ZPS_tsAplZdpActiveEpStoreRsp`, described in Section 8.2.3.19
    - `sSimpleDescStoreRsp` is a structure of the type `ZPS_tsAplZdpSimpleDescStoreRsp`, described in Section 8.2.3.18

- sRemoveNodeCacheRsp is a structure of the type ZPS_tsAplZdpRemoveNodeCacheRsp, described in Section 8.2.3.21
- sEndDeviceBindRsp is a structure of the type ZPS_tsAplZdpEndDeviceBindRsp, described in Section 8.2.3.22
- sBindRsp is a structure of the type ZPS_tsAplZdpBindRsp, described in Section 8.2.3.23
- sUnbindRsp is a structure of the type ZPS_tsAplZdpUnbindRsp, described in Section 8.2.3.24
- sReplaceDeviceRsp is a structure of the type ZPS_tsAplZdpReplaceDeviceRsp, described in Section 8.2.3.26
- sStoreBkupBindEntryRsp is a structure of the type ZPS_tsAplZdpStoreBkupBindEntryRsp, described in Section 8.2.2.27
- sRemoveBkupBindEntryRsp is a structure of the type ZPS_tsAplZdpRemoveBkupBindEntryRsp, described in Section 8.2.2.28
- sBackupSourceBindRsp is a structure of the type ZPS_tsAplZdpBackupSourceBindRsp, described in Section 8.2.3.31
- sMgmtLeaveRsp is a structure of the type ZPS_tsAplZdpMgmtLeaveRsp, described in Section 8.2.3.37
- sMgmtDirectJoinRsp is a structure of the type ZPS_tsAplZdpMgmtDirectJoinRsp, described in Section 8.2.3.38
- sPermitJoiningRsp is a structure of the type ZPS_tsAplZdpMgmtPermitJoiningRsp, described in Section 8.2.3.39
- sNodeDescRsp is a structure of the type ZPS_tsAplZdpNodeDescRsp, described in Section 8.2.3.3
- sPowerDescRsp is a structure of the type ZPS_tsAplZdpPowerDescRsp, described in Section 8.2.3.4
- sSimpleDescRsp is a structure of the type ZPS_tsAplZdpSimpleDescRsp, described in Section 8.2.3.5
- sNwkAddrRsp is a structure of the type ZPS_tsAplZdpNwkAddrRsp, described in Section 8.2.3.1
- sIeeeAddrRsp is a structure of the type ZPS_tsAplZdpIeeeAddrRsp, described in Section 8.2.3.2
- sUserDescConf is a structure of the type ZPS_tsAplZdpUserDescConf, described in Section 8.2.3.12
- sSystemServerDiscoveryRsp is a structure of the type ZPS_tsAplZdpSystemServerDiscoveryRsp, described in Section 8.2.3.13
- sPowerDescStoreRsp is a structure of the type ZPS_tsAplZdpPowerDescStoreRsp, described in Section 8.2.3.17
- sUserDescRsp is a structure of the type ZPS_tsAplZdpUserDescRsp, described in Section 8.2.3.8

- sActiveEpRsp is a structure of the type ZPS_tsAplZdpActiveEpRsp, described in Section 8.2.3.10
- sMatchDescRsp is a structure of the type ZPS_tsAplZdpMatchDescRsp, described in Section 8.2.3.9
- sComplexDescRsp is a structure of the type ZPS_tsAplZdpComplexDescRsp, described in Section 8.2.3.7
- sFindNodeCacheRsp is a structure of the type ZPS_tsAplZdpFindNodeCacheRsp, described in Section 8.2.3.20
- sExtendedSimpleDescRsp is a structure of the type ZPS_tsAplZdpExtendedSimpleDescRsp, described in Section 8.2.3.6
- sExtendedActiveEpRsp is a structure of the type ZPS_tsAplZdpExtendedActiveEpRsp, described in Section 8.2.3.11
- sBindRegisterRsp is a structure of the type ZPS_tsAplZdpBindRegisterRsp, described in Section 8.2.3.25
- sBackupBindTableRsp is a structure of the type ZPS_tsAplZdpBackupBindTableRsp, described in Section 8.2.3.29
- sRecoverBindTableRsp is a structure of the type ZPS_tsAplZdpRecoverBindTableRsp, described in Section 8.2.3.30
- sRecoverSourceBindRsp is a structure of the type ZPS_tsAplZdpRecoverSourceBindRsp, described in Section 8.2.3.32
- sMgmtNwkDiscRsp is a structure of the type ZPS_tsAplZdpMgmtNwkDiscRsp, described in Section 8.2.3.33
- sMgmtLqiRsp is a structure of the type ZPS_tsAplZdpMgmtLqiRsp, described in Section 8.2.3.34
- sRtgRsp is a structure of the type ZPS_tsAplZdpMgmtRtgRsp, described in Section 8.2.3.35
- sMgmtBindRsp is a structure of the type ZPS_tsAplZdpMgmtBindRsp, described in Section 8.2.3.36
- sMgmtCacheRsp is a structure of the type ZPS_tsAplZdpMgmtCacheRsp, described in Section 8.2.3.40
- sMgmtNwkUpdateNotify is a structure of the type ZPS_tsAplZdpMgmtNwkUpdateNotify, described in Section 8.2.3.41

- uLists is a union of the different arrays/tables which act as temporary storage for data elements used by the stack (and are therefore for internal use only)

7.2.3 Other Structures

This section describes various structures used by the AF API.

The structures are listed below, along with their page references.

7.2.3.1 ZPS_tsNwkNetworkDescr

This structure is used in an array element in the structure `ZPS_tsAfNwkDiscoveryEvent`, which is created as part of the ZPS_EVENT_NWK_DISCOVERY_COMPLETE event which reports the networks detected during a network discovery (see Section 7.2.2.9).

The `ZPS_tsNwkNetworkDescr` structure contains information on a detected network and is detailed below.

```
typedef struct
{
  uint64    u64ExtPanId;
  uint8     u8LogicalChan;
  uint8     u8StackProfile;
  uint8     u8ZigBeeVersion;
  uint8     u8PermitJoining;
  uint8     u8RouterCapacity;
  uint8     u8EndDeviceCapacity;
} ZPS_tsNwkNetworkDescr;
```

where:

- `u64ExtPanId` is the Extended PAN ID of the discovered network
- `u8LogicalChan` is the 2400-MHz channel on which the network was found
- `u8StackProfile` is the Stack Profile of the discovered network (0 - manufacturer-specific, 1 - ZigBee, 2 - ZigBee PRO, other values reserved) and is fixed at 2 for the NXP stack
- `u8ZigBeeVersion` is the ZigBee version of the discovered network
- `u8PermitJoining` indicates the number of detected nodes with ‘permit joining’ enabled (and therefore allowing nodes to join the network through them)
- `u8RouterCapacity` indicates the number of detected nodes that are allowing Routers to join the network through them

- u8EndDeviceCapacity indicates the number of detected nodes that are allowing End Devices to join the network through them

7.2.3.2 ZPS_tsNwkNlmeCfmEdScan

This structure is used by the structure ZPS_tsAfNwkEdScanConfEvent, which is created as part of the ZPS_EVENT_NWK_ED_SCAN event which reports the results of an ‘energy detect’ scan in the 2.4-GHz radio band.

The ZPS_tsNwkNlmeCfmEdScant structure is detailed below.

```
typedef struct
{
     uint8 u8Status;
     uint8 u8ResultListSize;
     uint8 au8EnergyDetect[ZPS_NWK_MAX_ED_RESULTS];
} ZPS_tsNwkNlmeCfmEdScan;
```

where

- u8Status is one of the status codes from the lower stack layers, detailed in Section 10.2.
- u8ResultListSize is the number of entries in the results list (see below)
- au8EnergyDetect[] is an array containing the list of results of the energy scan (8-bit values representing the detected energy levels in the channels) - there is one array element for each channel scanned, where element 0 is for the first channel scanned, element 1 is for the second channel scanned, etc.

7.2.3.3 ZPS_tsInterPanAddress

This structure is used to specify the destination for an inter-PAN transmission.

The ZPS_tsInterPanAddress structure is detailed below.

```
typedef struct
{
  enum {
      ZPS_E_AM_INTERPAN_GROUP = 0x01,
      ZPS_E_AM_INTERPAN_SHORT,
      ZPS_E_AM_INTERPAN_IEEE
  }eMode;
  uint16        u16PanId;
  ZPS_tuAddress uAddress;
} ZPS_tsInterPanAddress;
```

where:

- eMode is used to specify the type of destination address that will be used in the field uAddress below - one of the following enumerations must be specified:

  - ZPS_E_AM_INTERPAN_GROUP indicates that a 16-bit group address will be used to specify multiple target nodes in the destination network (the group address must be valid in the destination network)
  - ZPS_E_AM_INTERPAN_SHORT indicates that a 16-bit network/short address will be used to specify a single target node or a broadcast to all nodes in the destination network
  - ZPS_E_AM_INTERPAN_IEEE indicates that a 64-bit IEEE/MAC address will be used to specify a single target node in the destination network
- `u16PanId` is the PAN ID of the destination network - a value 0xFFFF can be used to specify a broadcast to all reachable ZigBee PRO networks
- `uAddress` is the address of the target node(s) in the destination network (the address type must be as specified above in the `eMode` field) - a value of 0xFFFF can be used to specify a broadcast to all nodes in the destination network(s)

7.2.3.4 ZPS_tsAfProfileDataReq

This structure is used to specify the transmission details for a data transmission submitted using the function **ZPS_eAplAfApsdeDataReq()**.

The `ZPS_tsAfProfileDataReq` structure is detailed below.

```
typedef struct {
    ZPS_tuAddress                 uDstAddr;
    uint16                        u16ClusterId;
    uint16                        u16ProfileId;
    uint8                         u8SrcEp;
    ZPS_teAplApsdeAddressMode     eDstAddrMode;
    uint8                         u8DstEp;
    ZPS_teAplAfSecurityMode       eSecurityMode;
    uint8                         u8Radius;
}ZPS_tsAfProfileDataReq;
```

where:

- `uDstAddr` is the address of the destination node for the transmission request (can be 16- or 64-bit, as specified by `eDstAddrMode`)
- `u16ClusterId` is the Cluster ID of the destination cluster
- `u16ProfileId` is the Profile ID of the destination application profile
- `u8SrcEp` is the source endpoint number (1-240) on the local node
- `eDstAddrMode` is the type of destination address, one of (also see Table 7 on page 194):
  - ZPS_E_ADDR_MODE_BOUND
    (no address needed for bound nodes)
  - ZPS_E_ADDR_MODE_GROUP
    (16-bit group address)

  - ZPS_E_ADDR_MODE_SHORT
    (16-bit network address)
  - ZPS_E_ADDR_MODE_IEEE
    (64-bit IEEE/MAC address)
- `u8DstEp` is the destination endpoint number (1-240) on the remote node
- `eSecurityMode` is the security mode for the data transfer, one of:
  - ZPS_E_APL_AF_UNSECURE
    (no security enabled)
  - ZPS_E_APL_AF_SECURE
    (Application-level security using link key and network key)
  - ZPS_E_APL_AF_SECURE_NWK
    (Network-level security using network key)
  - ZPS_E_APL_AF_SECURE | ZPS_E_APL_AF_EXT_NONCE
    (Application-level security using link key and network key with the extended NONCE included in the frame)
  - ZPS_E_APL_AF_WILD_PROFILE
    (May be combined with above flags using OR operator. Sends the message using the wild card profile (0xFFFF) instead of the profile in the associated Simple descriptor)
- `u8Radius` is the maximum number of hops permitted to the destination node (zero value specifies that default maximum is to be used)

7.2.3.5 tsBeaconFilterType

This structure contains the details of a beacon filter that can be introduced using the function **ZPS_bAppAddBeaconFilter()**.

The `tsBeaconFilterType` structure is detailed below.

```
typedef struct
{
    uint64    *pu64ExtendPanIdList;
    uint16    u16Panid;
    uint16    u16FilterMap;
    uint8     u8ListSize;
    uint8     u8Lqi;
    uint8     u8Depth;
} tsBeaconFilterType;
```

where:

- `pu64ExtendPanIdList` is a pointer to a list of 64-bit Extended PAN IDs (EPIDs) which acts as a blacklist or whitelist of networks, depending on the settings of bits 0 and 1 in the `u8FilterMap` bitmap:
  - If this is a blacklist, beacons from networks with EPIDs in the list will not be accepted

  - If this is a whitelist, only beacons from networks with EPIDs in the list will be accepted
- `u16Panid` is a 16-bit PAN ID on which beacons can be filtered
- `u8ListSize` is the number of Extended PAN IDs in the list pointed to by `pu64ExtendPanIdList`
- `u8Lqi` is the minimum LQI value (in the range 0 to 255) of an acceptable beacon (any beacon with LQI value less than this minimum will be filtered out) - if required, this field must be enabled through bit 2 in the `u8FilterMap` bitmap
- `u8Depth` is the tree depth of the neighbour device. A value of 0x00 indicates that the device is the ZigBee coordinator for the network.
- `u16FilterMap` is an 16-bit bitmap detailing the filtering requirements, as follows:

Bit	Enumeration	Description
0	BF_BITMAP_BLACKLIST(0x1)	If set, field `pu64ExtendPanIdList` points to a blacklist of networks
1	BF_BITMAP_WHITELIST (0x2)	If set, field `pu64ExtendPanIdList` points to a whitelist of networks
2	BF_BITMAP_LQI (0x4)	If set, beacons must be filtered according to LQI value using the minimum in field `u8Lqi`
3	BF_BITMAP_CAP_ENDDEVICE (0x8)	If set, beacons from nodes with capacity for End Device children can be accepted
4	BF_BITMAP_CAP_ROUTER (0x10)	If set, beacons from nodes with capacity for Router children can be accepted
5	BF_BITMAP_PERMIT_JOIN (0x20)	If set, beacons from nodes with 'permit joining' enabled can be accepted
6	BF_BITMAP_SHORT_PAN (0x40)	If set, beacons from nodes in a network with the PAN ID in `u16Panid` can be accepted
7	-	Reserved
8	BF_BITMAP_DEPTH	If set, beacons from nodes in a network with the depth in u8Depth. if it is set to 0xff - filters out any beacon which is not from the coordinator. Any other value of u8Depth, filters out beacons with greater than or equal to `u8Depth`

Note: Bits 0 and 1 must not both be set

Note: After each discovery or rejoin, the flags contained in the `u16FilterMap` field will be cleared while all other fields of this structure will remain intact.

7.2.3.6 ZPS_tsAplApsKeyDescriptorEntry

This structure contains a link key for secured communications with another node.

```
typedef struct
{
    uint32 u32OutgoingFrameCounter;
    uint16 u16ExtAddrLkup;
    uint8  au8LinkKey[ZPS_SEC_KEY_LENGTH];
} ZPS_tsAplApsKeyDescriptorEntry;
```

where:

- `u32OutgoingFrameCounter` is the outgoing frame counter value which is incremented on each transmission to a destination address below
- `u16ExtAddrLkup` is the index of the local look-up table entry that contains the IEEE/MAC address of either the Trust Centre or the target node
- `au8LinkKey[]` is an array containing the link key

7.2.3.7 ZPS_tsAftsStartParamsDistributed

This structure contains the start parameter values for a node in a distributed security network.

```
typedef struct
{
    uint64 u64ExtPanId;
    uint8 *pu8NwkKey;
    uint16 u16PanId;
    uint16 u16NwkAddr;
    uint8 u8KeyIndex;
    uint8 u8LogicalChannel;
    uint8 u8NwkupdateId;
} ZPS_tsAftsStartParamsDistributed;
```

where:

- `u64ExtPanId` is the Extended PAN ID of the distributed security network
- `pu8NwkKey` is a pointer to a location to receive the network key
- `u16PanId` is the PAN ID of the network
- `u16NwkAddr` is the network address of the local node
- `u8KeyIndex` is the sequence number required to identify the network key in the security set
- `u8LogicalChannel` is the number of the radio channel on which the network operates

- `u8NwkupdateId` is a unique byte value which is incremented when the network parameters are updated (and is therefore used to determine whether a receiving node has missed an update)

7.2.3.8 ZPS_tsAfFlashInfoSet

This structure contains information about the devices Flash memory sector used for the persistent storage of unique link keys on the Trust Centre, as enabled by the function **ZPS_vTcInitFlash()**.

```
typedef struct
{
  uint16    u16SectorSize;
  uint16    u16CredNodesCount;
  uint8     u8SectorSet;
} ZPS_tsAfFlashInfoSet;
```

where:

- `u16SectorSize` is the size, in bytes, of the Flash memory sector used to store the link keys
- `u16CredNodesCount` is the maximum number of nodes for which link keys can be stored in the Flash memory sector
- `u8SectorSet` is the number of the Flash memory sector used for this storage

Note: Care should be taken that this sector is set greater than the current flash usage of the image you are running. If this clashes with something else (image or user data) This will lead to flash corruption and the behaviour becomes non deterministic.

7.2.3.9 ZPS_TclkDescriptorEntry

This structure is used on the Trust Centre to hold information in RAM about a link key for a node, where this link key is held in persistent storage in devices Flash memory, as enabled by the function **ZPS_vTcInitFlash()**. If this feature is used, the application must allocate space for an array of these structures in RAM, with one structure for each potential node in the network.

```
typedef struct
{
   uint16    u16CredOffset;
   uint16    u16TclkRetries;
} ZPS_TclkDescriptorEntry;
```

where:

- `u16CredOffset` is the offset, in bytes, of the storage location for the node's link key in the relevant Flash memory sector
- `u16TclkRetries` is the number of retries that were attempted to negotiate the link key between the Trust Centre and the node

8. ZigBee Device Profile (ZDP) API

The chapter describes the resources of the ZigBee Device Profile (ZDP) API. This API is concerned with sending network requests (e.g. binding requests) and receiving responses. The API is defined in the header file **zps_apl_zdp.h**.

In this chapter:

- Section 8.1 details the ZDP API functions
- Section 8.2 details the ZDP API structures
- Section 8.3 describes the broadcast options when sending requests using the ZDP API functions

8.1 ZDP API Functions

The ZDP API functions are divided into the following categories:

- **Address Discovery** functions, described in Section 8.1.1
- **Service Discovery** functions, described in Section 8.1.2
- **Binding** functions, described in Section 8.1.3
- **Network Management Service** functions, described in Section 8.1.4
- **Response Data Extraction** function, described in Section 8.1.5

Common Parameters

All the ZDP API functions, except **ZPS_bAplZdpUnpackResponse()**, are concerned with sending out a request and all use a similar set of parameters. These parameters are described below, but more specific information is provided as part of the function descriptions:

- *hAPdu*: This is the unique handle of the APDU (Application Protocol Data Unit) instance for the request to be sent (see below).
- *uDstAddr*: This is the IEEE address or network address of the node to which the request will be sent (the parameter *bExtAddr* must be set according to the type of address used). For a broadcast, *uDstAddr* must be set to a special address, as described in Section 8.3.
- *bExtAddr*: This is a Boolean indicating the type of address specified in the parameter *uDstAddr* as a 64-bit IEEE address (TRUE) or 16-bit network address (FALSE).
- *pu8SeqNumber*: This is a pointer to the sequence number for the request - each request must have a unique sequence number to help determine the order in which requests were sent. On sending a request, the function automatically increments the sequence number for the next request.
- *u16ProfileId*: This is the identifier of the ZigBee application profile being used.
- *psZdpNwkAddrReq*: This is a pointer to a structure representing the request. The structure used is dependent on the specific function. The different request structures are detailed in Section 8.2.2.

APDUs for Requests and Responses

A request generated by this API is sent in an APDU (Application Protocol Data Unit). A local APDU instance for the request must first be allocated using the PDUM function **PDUM_hAPduAllocateAPduInstance()**. This function returns a handle for the APDU instance, which is subsequently used in the relevant ZDP API request function. Once the request has been successfully sent, the APDU instance is automatically de-allocated by the stack (there is no need for the application to de-allocate it).

Note: If the request is not successfully sent (the send function does not return ZPS_E_SUCCESS) then the APDU instance will not be automatically de-allocated and the application should de-allocate it using the PDUM function **PDUM_eAPduFreeAPduInstance()**.

When a response is subsequently received, the stack automatically allocates a local APDU instance and includes its handle in the notification event for the response. Once the response has been dealt with, the application must de-allocate the APDU instance using the function **PDUM_eAPduFreeAPduInstance()**.

8.1.1 Address Discovery Functions

The ZDP Address Discovery functions are concerned with obtaining addresses of nodes in the network.

The functions are listed below, along with their page references:

Note: Further addressing functions are provided in the ZDO API and are described in Section 6.1.3.

ZPS_eAplZdpNwkAddrRequest

ZPS_teStatus ZPS_eAplZdpNwkAddrRequest(
PDUM_thAPduInstance *hAPduInst*,
ZPS_tuAddress *uDstAddr*,
bool *bExtAddr*,
uint8 **pu8SeqNumber*,
ZPS_tsAplZdpNwkAddrReq **psZdpNwkAddrReq*);

Description

This function requests the 16-bit network address of the node with a particular 64-bit IEEE (MAC) address. The function sends out an NWK_addr_req request, which can be either unicast or broadcast, as follows:

- Unicast to another node, specified through *uDstAddr*, that will 'know' the required network address (this may be the parent of the node of interest or the Co-ordinator)
- Broadcast to the network, in which case *uDstAddr* must be set to the special network address 0xFFFF (see Section 8.3)

The IEEE address of the node of interest must be specified in the request, represented by the structure below (detailed further in Section 8.2.2.1).

```
typedef struct {
   uint64 u64IeeeAddr;
   uint8  u8RequestType;
   uint8  u8StartIndex;
} ZPS_tsAplZdpNwkAddrReq;
```

The required network address will be received in an NWK_addr_resp response, which should be collected using the function **ZQ_bZQueueReceive()** and stored in a structure of type `ZPS_tsAplZdpNwkAddrRsp` (detailed in Section 8.2.3.1). Note that this response can optionally contain the network addresses of the responding node's neighbours (this option is selected as part of the request through `u8RequestType`).

Parameters

hAPduInst	Handle of APDU instance in which request will be sent
uDstAddr	Address of destination node of request (can be 16- or 64-bit, as specified by *bExtAddr*)
bExtAddr	Type of destination address: TRUE: 64-bit IEEE (MAC) address FALSE: 16-bit network address
**pu8SeqNumber*	Pointer to sequence number of request
**psZdpNwkAddrReq*	Pointer to request (see above)

Returns

ZPS_E_SUCCESS (request successfully sent)

APS return codes, listed and described in Section 10.2.2

NWK return codes, listed and described in Section 10.2.3

MAC return codes, listed and described in Section 10.2.4

ZPS_eAplZdpIEEEAddrRequest

```
ZPS_teStatus ZPS_eAplZdpIeeeAddrRequest(
        PDUM_thAPduInstance hAPduInst,
        ZPS_tuAddress uDstAddr,
        bool bExtAddr,
        uint8 *pu8SeqNumber,
        ZPS_tsAplZdpIeeeAddrReq *psZdpIeeeAddrReq);
```

Description

This function requests the 64-bit IEEE (MAC) address of the node with a particular 16-bit network address. The function sends an IEEE_addr_req request to the relevant node, specified through *uDstAddr*.

The network address of the node of interest must also be specified in the request, represented by the structure below (detailed further in Section 8.2.2.2).

```
typedef struct {
   uint16 u16NwkAddrOfInterest;
   uint8  u8RequestType;
   uint8  u8StartIndex;
} ZPS_tsAplZdpIeeeAddrReq;
```

The required IEEE address will be received in an IEEE_addr_resp response, which should be collected using the function **ZQ_bZQueueReceive()** and stored in a structure of type `ZPS_tsAplZdpIeeeAddrRsp` (detailed in Section 8.2.3.2). Note that this response can optionally contain the IEEE addresses of the responding node's neighbours (this option is selected as part of the request through `u8RequestType`).

Parameters

hAPduInst	Handle of APDU instance in which request will be sent
uDstAddr	Network address of destination node of request (*bExtAddr* must be set to FALSE - see below)
bExtAddr	Type of destination address: TRUE: 64-bit IEEE (MAC) address FALSE: 16-bit network address
**pu8SeqNumber*	Pointer to sequence number of request
**psZdpIeeeAddrReq*	Pointer to request (see above)

Returns

ZPS_E_SUCCESS (request successfully sent)

APS return codes, listed and described in Section 10.2.2

NWK return codes, listed and described in Section 10.2.3

MAC return codes, listed and described in Section 10.2.4

ZPS_eAplZdpDeviceAnnceRequest

```
ZPS_teStatus ZPS_eAplZdpDeviceAnnceRequest(
    PDUM_thAPduInstance hAPduInst,
    uint8 *pu8SeqNumber,
    ZPS_tsAplZdpDeviceAnnceReq *psZdpDeviceAnnceReq);
```

Description

This function is used to notify other nodes that the local node has joined or rejoined the network. The function broadcasts a Device_annce announcement to the network and is normally automatically called by the ZDO when the local node joins or rejoins the network.

The IEEE (MAC) and allocated network addresses as well as the capabilities of the sending node must be specified in the announcement, represented by the structure below (detailed further in Section 8.2.2.3).

```
typedef struct {
   uint16 u16NwkAddr;
   uint64 u64IeeeAddr;
   uint8  u8Capability;
} ZPS_tsAplZdpDeviceAnnceReq;
```

On receiving this announcement, a network node will update any information it holds that relates to the supplied IEEE and network addresses:

- If it already holds the supplied IEEE address, it will update the corresponding network address with the supplied one (if necessary).
- If it already holds the supplied network address but with a different corresponding IEEE address, the latter will be marked as not having a valid corresponding network address.

Parameters

hAPduInst	Handle of APDU instance in which request will be sent
******pu8SeqNumber*	Pointer to sequence number of announcement
******psZdpDeviceAnnceReq*	Pointer to announcement (see above)

Returns

ZPS_E_SUCCESS (request successfully sent)

APS return codes, listed and described in Section 10.2.2

NWK return codes, listed and described in Section 10.2.3

MAC return codes, listed and described in Section 10.2.4

8.1.2 Service Discovery Functions

The ZDP Service Discovery functions are concerned with obtaining information about the nature and capabilities of a network node.

The functions are listed below, along with their page references:

ZPS_eAplZdpNodeDescRequest

ZPS_teStatus ZPS_eAplZdpNodeDescRequest(
PDUM_thAPduInstance *hAPduInst*,
ZPS_tuAddress *uDstAddr*,
bool *bExtAddr*,
uint8 **pu8SeqNumber*,
ZPS_tsAplZdpNodeDescReq **psZdpNodeDescReq*);

Description

This function requests the Node descriptor of the node with a particular network address. The function sends a Node_Desc_req request either to the relevant node or to another node that may hold the required information in its primary discovery cache.

The network address of the node of interest must be specified in the request, which is represented by the structure below (further detailed in Section 8.2.2.4).

```
typedef struct {
   uint16 u16NwkAddrOfInterest;
} ZPS_tsAplZdpNodeDescReq;
```

The required Node descriptor will be received in a Node_Desc_rsp response, which should be collected using the function **ZQ_bZQueueReceive()** and stored in a structure of type `ZPS_tsAplZdpNodeDescRsp` (detailed in Section 8.2.3.3).

Parameters

hAPduInst	Handle of APDU instance in which request will be sent
uDstAddr	Address of destination node of request (can be 16- or 64-bit, as specified by *bExtAddr*)
bExtAddr	Type of destination address: TRUE: 64-bit IEEE (MAC) address FALSE: 16-bit network address
**pu8SeqNumber*	Pointer to sequence number of request
**psZdpNodeDescReq*	Pointer to request (see above)

Returns

ZPS_E_SUCCESS (request successfully sent)

APS return codes, listed and described in Section 10.2.2

NWK return codes, listed and described in Section 10.2.3

MAC return codes, listed and described in Section 10.2.4

ZPS_eAplZdpPowerDescRequest

ZPS_teStatus ZPS_eAplZdpPowerDescRequest(
PDUM_thAPduInstance *hAPduInst*,
ZPS_tuAddress *uDstAddr*,
bool *bExtAddr*,
uint8 **pu8SeqNumber*,
ZPS_tsAplZdpPowerDescReq **psZdpPowerDescReq*);

Description

This function requests the Power descriptor of the node with a particular network address. The function sends a Power_Desc_req request either to the relevant node or to another node that may hold the required information in its primary discovery cache.

The network address of the node of interest must be specified in the request, which is represented by the structure below (further detailed in Section 8.2.2.5).

```
typedef struct {
   uint16 u16NwkAddrOfInterest;
} ZPS_tsAplZdpPowerDescReq;
```

The required Power descriptor will be received in a Power_Desc_rsp response, which should be collected using the function **ZQ_bZQueueReceive()** and stored in a structure of type `ZPS_tsAplZdpPowerDescRsp` (detailed in Section 8.2.3.4).

Parameters

hAPduInst	Handle of APDU instance in which request will be sent
uDstAddr	Address of destination node of request (can be 16- or 64-bit, as specified by *bExtAddr*)
bExtAddr	Type of destination address: TRUE: 64-bit IEEE (MAC) address FALSE: 16-bit network address
**pu8SeqNumber*	Pointer to sequence number of request
**psZdpPowerDescReq*	Pointer to request (see above)

Returns

ZPS_E_SUCCESS (request successfully sent)

APS return codes, listed and described in Section 10.2.2

NWK return codes, listed and described in Section 10.2.3

MAC return codes, listed and described in Section 10.2.4

ZPS_eAplZdpSimpleDescRequest

ZPS_teStatus ZPS_eAplZdpSimpleDescRequest(
PDUM_thAPduInstance *hAPduInst*,
ZPS_tuAddress *uDstAddr*,
bool *bExtAddr*,
uint8 **pu8SeqNumber*,
ZPS_tsAplZdpSimpleDescReq **psZdpSimpleDescReq*);

Description

This function requests the Simple descriptor for a specific endpoint on the node with a particular network address. The function sends a Simple_Desc_req request either to the relevant node or to another node that may hold the required information in its primary discovery cache.

The network address of the node of interest and the relevant endpoint on the node must be specified in the request, which is represented by the structure below (further detailed in Section 8.2.2.6).

```
typedef struct {
   uint16 u16NwkAddrOfInterest;
   uint8  u8EndPoint;
} ZPS_tsAplZdpSimpleDescReq;
```

The required Simple descriptor will be received in a Simple_Desc_rsp response, which should be collected using the function **ZQ_bZQueueReceive()** and stored in a structure of type `ZPS_tsAplZdpSimpleDescRsp` (detailed in Section 8.2.3.5).

Parameters

hAPduInst	Handle of APDU instance in which request will be sent
uDstAddr	Address of destination node of request (can be 16- or 64-bit, as specified by *bExtAddr*)
bExtAddr	Type of destination address: TRUE: 64-bit IEEE (MAC) address FALSE: 16-bit network address
**pu8SeqNumber*	Pointer to sequence number of request
**psZdpSimpleDescReq*	Pointer to request (see above)

Returns

ZPS_E_SUCCESS (request successfully sent)

APS return codes, listed and described in Section 10.2.2

NWK return codes, listed and described in Section 10.2.3

MAC return codes, listed and described in Section 10.2.4

ZPS_eAplZdpExtendedSimpleDescRequest

ZPS_teStatus ZPS_eAplZdpExtendedSimpleDescRequest(
PDUM_thAPduInstance *hAPduInst*,
ZPS_tuAddress *uDstAddr*,
bool *bExtAddr*,
uint8 **pu8SeqNumber*,
ZPS_tsAplZdpExtendedSimpleDescReq
psZdpExtendedSimpleDescReq*);**

Description

This function requests a cluster list for a specific endpoint on the node with a particular network address. The function should be called if the endpoint has more input or output clusters than could be included in the response to **ZPS_eAplZdpSimpleDescRequest()**. The function sends an Extended_Simple_Desc_req request either to the relevant node or to another node that may hold the required information in its primary discovery cache.

The network address of the node of interest and the relevant endpoint on the node must be specified in the request, which is represented by the structure below (further detailed in Section 8.2.2.7).

```
typedef struct {
   uint16 u16NwkAddr;
   uint8  u8EndPoint;
   uint8  u8StartIndex;
} ZPS_tsAplZdpExtendedSimpleDescReq;
```

This structure allows you to specify the first input/output cluster of interest in the endpoint's input and output cluster lists. Thus, this should normally be the cluster after the last one reported following a call to **ZPS_eAplZdpSimpleDescRequest()**.

The required cluster information will be received in a Extended_Simple_Desc_rsp response, which should be collected using the function **ZQ_bZQueueReceive()** and stored in a structure of type `ZPS_tsAplZdpExtendedSimpleDescRsp` (detailed in Section 8.2.3.6).

Parameters

hAPduInst	Handle of APDU instance in which request will be sent
uDstAddr	Address of destination node of request (can be 16- or 64-bit, as specified by *bExtAddr*)
bExtAddr	Type of destination address: TRUE: 64-bit IEEE (MAC) address FALSE: 16-bit network address
**pu8SeqNumber*	Pointer to sequence number of request
**psZdpExtendedSimpleDescReq*	Pointer to request (see above)

Returns

ZPS_E_SUCCESS (request successfully sent)

APS return codes, listed and described in Section 10.2.2

NWK return codes, listed and described in Section 10.2.3

MAC return codes, listed and described in Section 10.2.4

ZPS_eAplZdpComplexDescRequest

ZPS_teStatus ZPS_eAplZdpComplexDescRequest(
PDUM_thAPduInstance *hAPduInst*,
ZPS_tuAddress *uDstAddr*,
bool *bExtAddr*,
uint8 **pu8SeqNumber*,
ZPS_tsAplZdpComplexDescReq **psZdpComplexDescReq*);

Description

This function requests the Complex descriptor of the node with a particular network address. The function sends a Complex_Desc_req request either to the relevant node or to another node that may hold the required information in its primary discovery cache.

The network address of the node of interest must be specified in the request, which is represented by the structure below (further detailed in Section 8.2.2.8).

```
typedef struct {
    uint16 u16NwkAddrOfInterest;
} ZPS_tsAplZdpComplexDescReq;
```

The required Complex descriptor will be received in a Complex_Desc_rsp response, which should be collected using the function **ZQ_bZQueueReceive()** and stored in a structure of type `ZPS_tsAplZdpComplexDescRsp` (detailed in Section 8.2.3.7).

Parameters

hAPduInst	Handle of APDU instance in which request will be sent
uDstAddr	Address of destination node of request (can be 16- or 64-bit, as specified by *bExtAddr*)
bExtAddr	Type of destination address: TRUE: 64-bit IEEE (MAC) address FALSE: 16-bit network address
**pu8SeqNumber*	Pointer to sequence number of request
**psZdpComplexDescReq*	Pointer to request (see above)

Returns

ZPS_E_SUCCESS (request successfully sent)

APS return codes, listed and described in Section 10.2.2

NWK return codes, listed and described in Section 10.2.3

MAC return codes, listed and described in Section 10.2.4

ZPS_eAplZdpUserDescRequest

ZPS_teStatus ZPS_eAplZdpUserDescRequest(
PDUM_thAPduInstance *hAPduInst*,
ZPS_tuAddress *uDstAddr*,
bool *bExtAddr*,
uint8 **pu8SeqNumber*,
ZPS_tsAplZdpUserDescReq **psZdpUserDescReq*);

Description

This function requests the User descriptor of the node with a particular network address. The function sends a User_Desc_req request either to the relevant node or to another node that may hold the required information in its primary discovery cache.

Note: This function can only be used to access the User descriptor of a non-NXP device (which supports this descriptor), since the storage of a User descriptor on an NXP remove device is not supported.

The network address of the node of interest must be specified in the request, which is represented by the structure below (further detailed in Section 8.2.2.9).

```
typedef struct {
   uint16 u16NwkAddrOfInterest;
} ZPS_tsAplZdpUserDescReq;
```

The required User descriptor will be received in a User_Desc_rsp response, which should be collected using the function **ZQ_bZQueueReceive()** and stored in a structure of type `ZPS_tsAplZdpUserDescRsp` (detailed in Section 8.2.3.8).

Parameters

hAPduInst	Handle of APDU instance in which request will be sent
uDstAddr	Address of destination node of request (can be 16- or 64-bit, as specified by *bExtAddr*)
bExtAddr	Type of destination address: TRUE: 64-bit IEEE (MAC) address FALSE: 16-bit network address
**pu8SeqNumber*	Pointer to sequence number of request
**psZdpUserDescReq*	Pointer to request (see above)

Returns

ZPS_E_SUCCESS (request successfully sent)

APS return codes, listed and described in Section 10.2.2

NWK return codes, listed and described in Section 10.2.3

MAC return codes, listed and described in Section 10.2.4

ZPS_eAplZdpMatchDescRequest

```
ZPS_teStatus ZPS_eAplZdpMatchDescRequest(
        PDUM_thAPduInstance hAPduInst,
        ZPS_tuAddress uDstAddr,
        bool bExtAddr,
        uint8 *pu8SeqNumber,
        ZPS_tsAplZdpMatchDescReq *psZdpMatchDescReq);
```

Description

This function requests responses from network nodes with endpoints that match specified criteria in their Simple descriptors. More specifically, these criteria include: application profile, number of input clusters, number of output clusters, list of input clusters, list of output clusters. The function sends out a Match_Desc_req command, as a broadcast to all network nodes, or as a unicast to either a specific node of interest or another node that may hold the required information in its primary discovery cache. The wild card profile (0xFFFF) can be used to match any profile ID.

The request is represented by the structure below (further detailed in Section 8.2.2.10).

```
typedef struct {
   uint16 u16NwkAddrOfInterest;
   uint16 u16ProfileId;
   /* rest of message is variable length */
   uint8  u8NumInClusters;
   uint16* pu16InClusterList;
   uint8  u8NumOutClusters;
   uint16* pu16OutClusterList;
} ZPS_tsAplZdpMatchDescReq;
```

A node with matching endpoint criteria will respond with a Match_Desc_rsp response, which should be collected using the function **ZQ_bZQueueReceive()** and stored in a structure of type `ZPS_tsAplZdpMatchDescRsp` (detailed in Section 8.2.3.9).

Parameters

hAPduInst	Handle of APDU instance in which request will be sent
uDstAddr	Address of destination node of request (can be 16- or 64-bit, as specified by *bExtAddr*)
bExtAddr	Type of destination address: TRUE: 64-bit IEEE (MAC) address FALSE: 16-bit network address
**pu8SeqNumber*	Pointer to sequence number of request
**psZdpMatchDescReq*	Pointer to request (see above)

Returns

ZPS_E_SUCCESS (request successfully sent)

APS return codes, listed and described in Section 10.2.2

NWK return codes, listed and described in Section 10.2.3

MAC return codes, listed and described in Section 10.2.4

ZPS_eAplZdpActiveEpRequest

```
ZPS_teStatus ZPS_eAplZdpActiveEpRequest(
        PDUM_thAPduInstance hAPduInst,
        ZPS_tuAddress uDstAddr,
        bool bExtAddr,
        uint8 *pu8SeqNumber,
        ZPS_tsAplZdpActiveEpReq *psZdpActiveEpReq);
```

Description

This function requests a list of the active endpoints on a remote node. The function sends an Active_EP_req request either to the relevant node or to another node that may hold the required information in its primary discovery cache.

The network address of the node of interest must be specified in the request, which is represented by the structure below (further detailed in Section 8.2.2.11).

```
typedef struct {
   uint16 u16NwkAddrOfInterest;
} ZPS_tsAplZdpActiveEpReq;
```

The endpoint list will be received in an Active_EP_rsp response, which should be collected using the function **ZQ_bZQueueReceive()** and stored in a structure of type `ZPS_tsAplZdpActiveEpRsp` (detailed in Section 8.2.3.10).

Parameters

hAPduInst	Handle of APDU instance in which request will be sent
uDstAddr	Address of destination node of request (can be 16- or 64-bit, as specified by *bExtAddr*)
bExtAddr	Type of destination address: TRUE: 64-bit IEEE (MAC) address FALSE: 16-bit network address
**pu8SeqNumber*	Pointer to sequence number of request
**psZdpActiveEpReq*	Pointer to request (see above)

Returns

ZPS_E_SUCCESS (request successfully sent)

APS return codes, listed and described in Section 10.2.2

NWK return codes, listed and described in Section 10.2.3

MAC return codes, listed and described in Section 10.2.4

ZPS_eAplZdpExtendedActiveEpRequest

ZPS_teStatus ZPS_eAplZdpExtendedActiveEpRequest(
PDUM_thAPduInstance *hAPduInst*,
ZPS_tuAddress *uDstAddr*,
bool *bExtAddr*,
uint8 **pu8SeqNumber*,
ZPS_tsAplZdpExtendedActiveEpReq
psZdpExtendedActiveEpReq*);**

Description

This function requests a list of the active endpoints on a remote node. The function should be called if the node has more active endpoints than could be included in a response to **ZPS_eAplZdpActiveEpRequest()**. The function sends an Extended_Active_EP_req request either to the relevant node or to another node that may hold the required information in its primary discovery cache.

The network address of the node of interest must be specified in the request, which is represented by the structure below (further detailed in Section 8.2.2.12).

```
typedef struct {
   uint16 u16NwkAddr;
   uint8  u8StartIndex;
} ZPS_tsAplZdpExtendedActiveEpReq;
```

This structure allows you to specify the first endpoint of interest for the request.

The endpoint list will be received in an Extended_Active_EP_rsp response, which should be collected using the function **ZQ_bZQueueReceive()** and stored in a structure of type `ZPS_tsAplZdpExtendedActiveEpRsp` (detailed in Section 8.2.3.11).

Parameters

hAPduInst	Handle of APDU instance in which request will be sent
uDstAddr	Address of destination node of request (can be 16- or 64-bit, as specified by *bExtAddr*)
bExtAddr	Type of destination address: TRUE: 64-bit IEEE (MAC) address FALSE: 16-bit network address
******pu8SeqNumber*	Pointer to sequence number of request
******psZdpActiveEpReq*	Pointer to request (see above)

Returns

ZPS_E_SUCCESS (request successfully sent)

APS return codes, listed and described in Section 10.2.2

NWK return codes, listed and described in Section 10.2.3

MAC return codes, listed and described in Section 10.2.4

ZPS_eAplZdpUserDescSetRequest

ZPS_teStatus ZPS_eAplZdpUserDescSetRequest(
PDUM_thAPduInstance *hAPduInst*,
ZPS_tuAddress *uDstAddr*,
bool *bExtAddr*,
uint8 **pu8SeqNumber*,
ZPS_tsAplZdpUserDescSet **psZdpUserDescSetReq*);

Description

This function can be used to configure the User descriptor on a remote node. The function sends a User_Desc_set request either to the remote node or to another node that may hold the relevant User descriptor in its primary discovery cache.

Note: This function can only be used to access the User descriptor of a non-NXP device (which supports this descriptor), since the storage of a User descriptor on an NXP device is not supported.

The network address of the node of interest as well as the required modifications must be specified in the request, which is represented by the structure below (further detailed in Section 8.2.2.13).

```
typedef struct {
  uint16 u16NwkAddrOfInterest;
  uint8  u8Length;
  char szUserDescriptor[ZPS_ZDP_LENGTH_OF_USER_DESC];
} ZPS_tsAplZdpUserDescSet;
```

If the specified User descriptor was successfully modified, a User_Desc_conf response will be received. This response should be collected by the application task using the function **ZQ_bZQueueReceive()** and stored in a structure of type `ZPS_tsAplZdpUserDescConf` (detailed in Section 8.2.3.12).

Parameters

hAPduInst	Handle of APDU instance in which request will be sent
uDstAddr	Address of destination node of request (can be 16- or 64-bit, as specified by *bExtAddr*)
bExtAddr	Type of destination address: TRUE: 64-bit IEEE (MAC) address FALSE: 16-bit network address
**pu8SeqNumber*	Pointer to sequence number of request
**psZdpUserDescSetReq*	Pointer to request (see above)

Returns

ZPS_E_SUCCESS (request successfully sent)

APS return codes, listed and described in Section 10.2.2

NWK return codes, listed and described in Section 10.2.3

MAC return codes, listed and described in Section 10.2.4

ZPS_eAplZdpSystemServerDiscoveryRequest

```
ZPS_teStatus ZPS_eAplZdpSystemServerDiscoveryRequest(
  PDUM_thAPduInstance hAPduInst,
  uint8 *pu8SeqNumber,
  ZPS_tsAplZdpSystemServerDiscoveryReq
                *psZdpSystemServerDiscoveryReq);
```

Description

This function can be used to request information on the available servers hosted by remote nodes (Primary or Backup Trust Centre, Primary or Backup Binding Table Cache, Primary or Backup Discovery Cache, Network Manager). The function broadcasts a System_Server_Discovery_req request to all network nodes.

The required servers must be specified by means of a bitmask in the request, which is represented by the structure below (further detailed in Section 8.2.2.14).

```
typedef struct {
    uint16 u16ServerMask;
} ZPS_tsAplZdpSystemServerDiscoveryReq;
```

A remote node will reply with a System_Server_Discovery_rsp response, indicating which of the requested servers are implemented. This response should be collected using the function **ZQ_bZQueueReceive()** and stored in a structure of type `ZPS_tsAplZdpSystemServerDiscoveryRsp` (detailed in Section 8.2.3.13).

Parameters

hAPduInst	Handle of APDU instance in which request will be sent
**pu8SeqNumber*	Pointer to sequence number of request
**psZdpSystemServerDiscoveryReq*	Pointer to request (see above)

Returns

ZPS_E_SUCCESS (request successfully sent)

APS return codes, listed and described in Section 10.2.2

NWK return codes, listed and described in Section 10.2.3

MAC return codes, listed and described in Section 10.2.4

ZPS_eAplZdpDiscoveryCacheRequest

ZPS_teStatus ZPS_eAplZdpDiscoveryCacheRequest(
PDUM_thAPduInstance *hAPduInst*,
uint8 **pu8SeqNumber*,
ZPS_tsAplZdpDiscoveryCacheReq
psZdpDiscoveryCacheReq*);**

Description

This function is used to discover which nodes in the network have a primary discovery cache - that is, a bank of information about other nodes in the network. The function broadcasts a Discovery_Cache_req request to the network.

The request includes the network and IEEE addresses of the sending device, and is represented by the structure below (further detailed in Section 8.2.2.15).

```
typedef struct {
    uint16 u16NwkAddr;
    uint64 u64IeeeAddr;
} ZPS_tsAplZdpDiscoveryCacheReq;
```

A node with a primary discovery cache replies with a Discovery_Cache_rsp response, which should be collected using the function **ZQ_bZQueueReceive()** and stored in a structure of type `ZPS_tsAplZdpDiscoveryCacheRsp` (detailed in Section 8.2.3.14).

Parameters

hAPduInst	Handle of APDU instance in which request will be sent
**pu8SeqNumber*	Pointer to sequence number of request
**psZdpDiscoveryCacheReq*	Pointer to request (see above)

Returns

ZPS_E_SUCCESS (request successfully sent)

APS return codes, listed and described in Section 10.2.2

NWK return codes, listed and described in Section 10.2.3

MAC return codes, listed and described in Section 10.2.4

ZPS_eAplZdpDiscoveryStoreRequest

```
ZPS_teStatus ZPS_eAplZdpDiscoveryStoreRequest(
  PDUM_thAPduInstance hAPduInst,
  ZPS_tuAddress uDstAddr,
  bool bExtAddr,
  uint8 *pu8SeqNumber,
  ZPS_tsAplZdpDiscoveryStoreReq
                    *psZdpDiscoveryStoreReq);
```

Description

This function can be called on an End Device to request a remote node to reserve memory space to store the local node's 'discovery information'. To do this, the remote node must contain a primary discovery cache. The 'discovery information' includes the local node's IEEE address, network address, Node descriptor, Power descriptor, Simple descriptor and number of active endpoints. The function sends a Discovery_store_req request to the remote node.

This request includes the network and IEEE addresses of the sending node as well as the amount of storage space (in bytes) needed to store the information. The request is represented by the structure below (further detailed in Section 8.2.2.16).

```
typedef struct {
   uint16 u16NwkAddr;
   uint64 u64IeeeAddr;
   uint8  u8NodeDescSize;
   uint8  u8PowerDescSize;
   uint8  u8ActiveEpSize;
   uint8  u8SimpleDescCount;
   /* Rest of message is variable length */
   uint8* pu8SimpleDescSizeList;
} ZPS_tsAplZdpDiscoveryStoreReq;
```

On receiving this request, the remote node will first check whether it has a primary discovery cache. If this is the case, it will check whether it has storage space in the cache for the new discovery information. If the space is available, it will be reserved until the information is later uploaded from the local node.

The node replies with a Discovery_store_rsp response, which should be collected using the function **ZQ_bZQueueReceive()** and stored in a structure of type `ZPS_tsAplZdpDiscoveryStoreRsp` (detailed in Section 8.2.3.15).

Parameters

hAPduInst,	Handle of APDU instance in which request will be sent
uDstAddr	Address of destination node of request (can be 16- or 64-bit, as specified by *bExtAddr*)
bExtAddr	Type of destination address: TRUE: 64-bit IEEE (MAC) address FALSE: 16-bit network address
******pu8SeqNumber*	Pointer to sequence number of request
******psZdpDiscoveryStoreReq*	Pointer to request (see above)

Returns

ZPS_E_SUCCESS (request successfully sent)

APS return codes, listed and described in Section 10.2.2

NWK return codes, listed and described in Section 10.2.3

MAC return codes, listed and described in Section 10.2.4

ZPS_eAplZdpNodeDescStoreRequest

ZPS_teStatus ZPS_eAplZdpNodeDescStoreRequest(
PDUM_thAPduInstance *hAPduInst*,
ZPS_tuAddress *uDstAddr*,
bool *bExtAddr*,
uint8 **pu8SeqNumber*,
ZPS_tsAplZdpNodeDescStoreReq
psZdpNodeDescStoreReq*);**

Description

This function can be called on an End Device to upload the local node's Node descriptor for storage in the primary discovery cache on a remote node. The function sends a Node_Desc_store_req command to the remote node.

This request includes the network and IEEE addresses of the sending node as well as the Node descriptor to store. The request is represented by the structure below (further detailed in Section 8.2.2.17).

```
typedef struct {
    uint16 u16NwkAddr;
    uint64 u64IeeeAddr;
    /* Rest of message is variable length */
    ZPS_tsAplZdpNodeDescriptor sNodeDescriptor;
} ZPS_tsAplZdpNodeDescStoreReq;
```

On receiving the request, the remote node will first check whether it has a primary discovery cache. If this is the case, it will check whether it has previously reserved storage space in its cache for the local node. If it has, it will store the Node descriptor in its cache.

The node replies with a Node_Desc_store_rsp response, which should be collected using the function **ZQ_bZQueueReceive()** and stored in a structure of type `ZPS_tsAplZdpNodeDescStoreRsp` (detailed in Section 8.2.3.16).

Note: This function should only be called if storage space for the local node's 'discovery information' has previously been reserved on the remote node following a call to **ZPS_eAplZdpDiscoveryStoreRequest()**.

Parameters

hAPduInst	Handle of APDU instance in which request will be sent
uDstAddr	Address of destination node of request (can be 16- or 64-bit, as specified by *bExtAddr*)
bExtAddr	Type of destination address: TRUE: 64-bit IEEE (MAC) address FALSE: 16-bit network address
******pu8SeqNumber*	Pointer to sequence number of request
******psZdpNodeDescStoreReq*	Pointer to request (see above)

Returns

ZPS_E_SUCCESS (request successfully sent)

APS return codes, listed and described in Section 10.2.2

NWK return codes, listed and described in Section 10.2.3

MAC return codes, listed and described in Section 10.2.4

ZPS_eAplZdpPowerDescStoreRequest

```
ZPS_teStatus ZPS_eAplZdpPowerDescStoreRequest(
  PDUM_thAPduInstance hAPduInst,
  ZPS_tuAddress uDstAddr,
  bool bExtAddr,
  uint8 *pu8SeqNumber,
  ZPS_tsAplZdpPowerDescStoreReq
                    *psZdpPowerDescStoreReq);
```

Description

This function can be called on an End Device to upload the local node's Power descriptor for storage in the primary discovery cache on a remote node. The function sends a Power_Desc_store_req request to the remote node.

This request includes the network and IEEE addresses of the sending node as well as the Power descriptor to store. The request is represented by the structure below (further detailed in Section 8.2.2.18).

```
typedef struct {
   uint16 u16NwkAddr;
   uint64 u64IeeeAddr;
   /* Rest of message is variable length */
   ZPS_tsAplZdpNodePowerDescriptor sPowerDescriptor;
} ZPS_tsAplZdpPowerDescStoreReq;
```

On receiving the request, the remote node will first check whether it has a primary discovery cache. If this is the case, it will check whether it has previously reserved storage space in its cache for the local node. If it has, it will store the Power descriptor in its cache.

The node replies with a Power_Desc_store_rsp response, which should be collected using the function **ZQ_bZQueueReceive()** and stored in a structure of type `ZPS_tsAplZdpPowerDescStoreRsp` (detailed in Section 8.2.3.17).

> **Note:** This function should only be called if storage space for the local node's 'discovery information' has previously been reserved on the remote node following a call to **ZPS_eAplZdpDiscoveryStoreRequest()**.

Parameters

hAPduInst	Handle of APDU instance in which request will be sent
uDstAddr	Address of destination node of request (can be 16- or 64-bit, as specified by *bExtAddr*)
bExtAddr	Type of destination address: TRUE: 64-bit IEEE (MAC) address FALSE: 16-bit network address
***pu8SeqNumber*	Pointer to sequence number of request
***psZdpPowerDescStoreReq*	Pointer to request (see above)

Returns

ZPS_E_SUCCESS (request successfully sent)

APS return codes, listed and described in Section 10.2.2

NWK return codes, listed and described in Section 10.2.3

MAC return codes, listed and described in Section 10.2.4

ZPS_eAplZdpSimpleDescStoreRequest

ZPS_teStatus ZPS_eAplZdpSimpleDescStoreRequest(
PDUM_thAPduInstance *hAPduInst*,
ZPS_tuAddress *uDstAddr*,
bool *bExtAddr*,
uint8 **pu8SeqNumber*,
ZPS_tsAplZdpSimpleDescStoreReq
psZdpSimpleDescStoreReq*);**

Description

This function can be called on an End Device to upload a Simple descriptor from the local node for storage in the primary discovery cache on the specified remote node. The Simple descriptor for each endpoint on the local node must be uploaded separately using this function. The function sends a Simple_Desc_store_req request to the remote node.

This request includes the network and IEEE addresses of the sending node as well as the Simple descriptor to store. The request is represented by the structure below (further detailed in Section 8.2.2.19).

```
typedef struct {
   uint16 u16NwkAddr;
   uint64 u64IeeeAddr;
   uint8  u8Length;
   /* Rest of message is variable length */
   ZPS_tsAplZdpSimpleDescType sSimpleDescriptor;
} ZPS_tsAplZdpSimpleDescStoreReq;
```

On receiving the request, the remote node will first check whether it has a primary discovery cache. If this is the case, it will check whether it has previously reserved storage space in its cache for the local node. If it has, it will store the Simple descriptor in its cache.

The node replies with a Simple_Desc_store_rsp response, which should be collected using the function **ZQ_bZQueueReceive()** and stored in a structure of type `ZPS_tsAplZdpSimpleDescStoreRsp` (detailed in Section 8.2.3.18).

Note: This function should only be called if storage space for the local node's 'discovery information' has previously been reserved on the remote node following a call to **ZPS_eAplZdpDiscoveryStoreRequest()**.

Parameters

hAPduInst	Handle of APDU instance in which request will be sent
uDstAddr	Address of destination node of request (can be 16- or 64-bit, as specified by *bExtAddr*)
bExtAddr	Type of destination address: TRUE: 64-bit IEEE (MAC) address FALSE: 16-bit network address
***pu8SeqNumber*	Pointer to sequence number of request
***psZdpSimpleDescStoreReq*	Pointer to request (see above)

Returns

ZPS_E_SUCCESS (request successfully sent)

APS return codes, listed and described in Section 10.2.2

NWK return codes, listed and described in Section 10.2.3

MAC return codes, listed and described in Section 10.2.4

ZPS_eAplZdpActiveEpStoreRequest

ZPS_teStatus ZPS_eAplZdpActiveEpStoreRequest(
PDUM_thAPduInstance *hAPduInst*,
ZPS_tuAddress *uDstAddr*,
bool *bExtAddr*,
uint8 ******pu8SeqNumber*,
ZPS_tsAplZdpActiveEpStoreReq
******psZdpActiveEpStoreReq***);**

Description

This function can be called on an End Device to upload a list of its active endpoints for storage in the primary discovery cache on a remote node. The function sends an Active_EP_store_req command to the remote node.

This request includes the network and IEEE addresses of the sending node as well as the list of active endpoints to store. The request is represented by the structure below (further detailed in Section 8.2.2.20).

```
typedef struct {
   uint16 u16NwkAddr;
   uint64 u64IeeeAddr;
   uint8  u8ActiveEPCount;
   /* Rest of message is variable length */
   uint8* pu8ActiveEpList;
} ZPS_tsAplZdpActiveEpStoreReq;
```

On receiving the request, the remote node will first check whether it has a primary discovery cache. If this is the case, it will check whether it has previously reserved storage space in its cache for the local node. If it has, it will store the list of active endpoints in its cache.

The node replies with an Active_EP_store_rsp response, which should be collected using the function **ZQ_bZQueueReceive()** and stored in a structure of type `ZPS_tsAplZdpActiveEpStoreRsp` (detailed in Section 8.2.3.19).

Note: This function should only be called if storage space for the local node's 'discovery information' has previously been reserved on the remote node following a call to **ZPS_eAplZdpDiscoveryStoreRequest()**.

Parameters

hAPduInst	Handle of APDU instance in which request will be sent
uDstAddr	Address of destination node of request (can be 16- or 64-bit, as specified by *bExtAddr*)
bExtAddr	Type of destination address: TRUE: 64-bit IEEE (MAC) address FALSE: 16-bit network address
**pu8SeqNumber*	Pointer to sequence number of request
**psZdpActiveEpStoreReq*	Pointer to request (see above)

Returns

ZPS_E_SUCCESS (request successfully sent)

APS return codes, listed and described in Section 10.2.2

NWK return codes, listed and described in Section 10.2.3

MAC return codes, listed and described in Section 10.2.4

ZPS_eAplZdpFindNodeCacheRequest

```
ZPS_teStatus ZPS_eAplZdpFindNodeCacheRequest(
  PDUM_thAPduInstance hAPduInst,
  uint8 *pu8SeqNumber,
  ZPS_tsAplZdpFindNodeCacheReq
              *psZdpFindNodeCacheReq);
```

Description

This function can be used to search for nodes in the network that hold 'discovery information' about a particular node. The function broadcasts a Find_node_cache_req request to the network.

This request includes the network and IEEE addresses of the node of interest. The request is represented by the structure below (further detailed in Section 8.2.2.21).

```
typedef struct {
   uint16 u16NwkAddr;
   uint64 u64IeeeAddr;
} ZPS_tsAplZdpFindNodeCacheReq;
```

On receiving the request, a remote node will first check whether it has a primary discovery cache, or is the specified node itself. If either is the case, it will check whether it holds the required information and, if this is the case, will reply with a Find_node_cache_rsp response. This response should be collected using the function **ZQ_bZQueueReceive()** and stored in a structure of type `ZPS_tsAplZdpFindNodeCacheRsp` (detailed in Section 8.2.3.20).

Only nodes that hold the required information will respond.

Parameters

hAPduInst	Handle of APDU instance in which request will be sent
**pu8SeqNumber*	Pointer to sequence number of request
**psZdpFindNodeCacheReq*	Pointer to request (see above)

Returns

ZPS_E_SUCCESS (request successfully sent)

APS return codes, listed and described in Section 10.2.2

NWK return codes, listed and described in Section 10.2.3

MAC return codes, listed and described in Section 10.2.4

ZPS_eAplZdpRemoveNodeCacheRequest

```
ZPS_teStatus ZPS_eAplZdpRemoveNodeCacheRequest(
  PDUM_thAPduInstance hAPduInst,
  ZPS_tuAddress uDstAddr,
  bool bExtAddr,
  uint8 *pu8SeqNumber,
  ZPS_tsAplZdpRemoveNodeCacheReq
              *psZdpRemoveNodeCacheReq);
```

Description

This function requests a Primary Discovery Cache node to remove from its cache all 'discovery information' relating to a particular End Device. The function sends a Remove_node_cache_req request to the Primary Discovery Cache node.

The effect of a successful request is to remove the relevant 'discovery information' and free the corresponding storage space in the cache previously reserved by **ZPS_eAplZdpDiscoveryStoreRequest()** (which may have been called from another node in the network).

This request includes the network and IEEE addresses of the End Device whose 'discovery information' is to be removed. The request is represented by the structure below (further detailed in Section 8.2.2.22).

```
typedef struct {
   uint16 u16NwkAddr;
   uint64 u64IeeeAddr;
} ZPS_tsAplZdpRemoveNodeCacheReq;
```

On receiving the request, the remote node will first check whether it has a primary discovery cache. If this is the case, it will check whether it has previously received and implemented a Discovery_store_req request for the specified End Device, resulting from a call to **ZPS_eAplZdpDiscoveryStoreRequest()**. If it has, it will delete the relevant data and unreserve the corresponding part of the cache.

The node replies with a Remove_node_cache_rsp response, which should be collected using the function **ZQ_bZQueueReceive()** and stored in a structure of type `ZPS_tsAplZdpRemoveNodeCacheRsp` (detailed in Section 8.2.3.21).

Parameters

hAPduInst	Handle of APDU instance in which request will be sent
uDstAddr	Address of destination node of request (can be 16- or 64-bit, as specified by *bExtAddr*)
bExtAddr	Type of destination address: TRUE: 64-bit IEEE (MAC) address FALSE: 16-bit network address
******pu8SeqNumber*	Pointer to sequence number of request
******psZdpRemoveNodeCacheReq*	Pointer to request (see above)

Returns

ZPS_E_SUCCESS (request successfully sent)

APS return codes, listed and described in Section 10.2.2

NWK return codes, listed and described in Section 10.2.3

MAC return codes, listed and described in Section 10.2.4

8.1.3 Binding Functions

The ZDP Binding functions are concerned with binding nodes together, to aid communication between them, and managing binding tables.

The functions are listed below, along with their page references:

Note 1: Some of the above binding functions cannot be used to send requests to nodes that run the NXP ZigBee PRO stack. They are supplied in the NXP ZDP API in order to facilitate interoperability with nodes based on non-NXP software which supports the corresponding requests. If applicable, this restriction is noted in the function description.

Note 2: Further binding functions are provided in the ZDO API and are described in Section 6.1.1.

ZPS_eAplZdpEndDeviceBindRequest

ZPS_teStatus ZPS_eAplZdpEndDeviceBindRequest(
PDUM_thAPduInstance *hAPduInst*,
uint8 **pu8SeqNumber*,
ZPS_tsAplZdpEndDeviceBindReq
******psZdpEndDeviceBindReq***);**

Description

This function sends a binding request to the Co-ordinator in order to bind an endpoint on the local node to an endpoint on a remote node (these nodes can be End Devices or Routers). The function should normally be invoked as the result of a user action on the local node, such as pressing a button. The function sends an End_Device_Bind_req request to the Co-ordinator.

This request includes details of the source node, endpoint and clusters. The request is represented by the structure below (further detailed in Section 8.2.2.23).

```
typedef struct {
   uint16 u16BindingTarget;
   uint64 u64SrcIeeeAddress;
   uint8  u8SrcEndpoint;
   uint16 u16ProfileId;
   /* Rest of message is variable length */
   uint8 u8NumInClusters;
   uint16 *pu16InClusterList;
   uint8 u8NumOutClusters;
   uint16 *pu16OutClusterList;
} ZPS_tsAplZdpEndDeviceBindReq;
```

On receiving the request, the Co-ordinator waits (for a pre-defined timeout period) for another binding request, from a different node, so that it can pair the requests and bind the endpoints. In order to bind the endpoints, their application profile IDs must match, and they must have compatible clusters in their input and output cluster lists.

The Co-ordinator replies to a binding request with an End_Device_Bind_rsp response, which should be collected on the requesting node using the function **ZQ_bZQueueReceive()** and stored in a structure of type `ZPS_tsAplZdpEndDeviceBindRsp` (detailed in Section 8.2.3.22).

The stack will automatically update the Binding tables on the two End Devices (following further bind requests from the Co-ordinator) and an ZPS_EVENT_ZDO_BIND event will be generated on the End Devices to signal these updates.

Parameters

hAPduInst	Handle of APDU instance in which request will be sent
***pu8SeqNumber**	Pointer to sequence number of request
***psZdpEndDeviceBindReq**	Pointer to request (see above)

Returns

ZPS_E_SUCCESS (request successfully sent)

APS return codes, listed and described in Section 10.2.2

NWK return codes, listed and described in Section 10.2.3

MAC return codes, listed and described in Section 10.2.4

ZPS_eAplZdpBindUnbindRequest

ZPS_teStatus ZPS_eAplZdpBindUnbindRequest(
PDUM_thAPduInstance *hAPduInst*,
ZPS_tuAddress *uDstAddr*,
bool *bExtAddr*,
uint8 **pu8SeqNumber*,
bool *bBindReq*,
ZPS_tsAplZdpBindUnbindReq **psZdpBindReq*);

Description

This function sends a binding or unbinding request (as specified) to a remote node which hosts a binding table. The function requests a modification of the binding table in order to bind or unbind two endpoints of nodes in the network. The nodes to be bound/unbound may be different from the node sending the request and the node receiving the request. The latter must be either a node with a primary binding table cache or the source node for the binding. This function could typically be used in a commissioning application to configure bindings between nodes during system set-up.

The function sends a Bind_req or Unbind_req request to the remote node which hosts the binding table to be modified. This request includes details of the source node and endpoint, and the target node and endpoint for the binding. The request is represented by the structure below (further detailed in Section 8.2.2.24).

```
typedef struct {
    uint64 u64SrcAddress;
    uint8  u8SrcEndpoint;
    uint16 u16ClusterId;
    uint8  u8DstAddrMode;
    union {
        struct {
            uint16 u16DstAddress;
        } sShort;
        struct {
            uint64 u64DstAddress;
            uint8  u8DstEndPoint;
        } sExtended;
     } uAddressField;
 } ZPS_tsAplZdpBindUnbindReq;
```

On receiving the request, the remote node adds or removes the relevant entry in its binding table and locally generates the event ZPS_EVENT_ZDO_BIND or ZPS_EVENT_ZDO_UNBIND, as appropriate, to signal the relevant update.

If the remote node holds a primary binding table cache, it will check whether the source node for the binding holds a table of its own source bindings (see the description of **ZPS_eAplZdpBindRegisterRequest()**) and, if so, automatically requests an update of this table. A node with a primary binding table cache will also request an update of the back-up cache, if one exists.

The remote node replies with a Bind_rsp or Unbind_rsp response, which should be collected using the function **ZQ_bZQueueReceive()** and stored in a structure of type `ZPS_tsAplZdpBindRsp` (detailed in Section 8.2.3.23) or `ZPS_tsAplZdpUnbindRsp` (detailed in Section 8.2.3.24).

Parameters

hAPduInst	Handle of APDU instance in which request will be sent
uDstAddr	Address of destination node of request (can be 16- or 64-bit, as specified by *bExtAddr*)
bExtAddr	Type of destination address: TRUE: 64-bit IEEE (MAC) address FALSE: 16-bit network address
**pu8SeqNumber*	Pointer to sequence number of request
bBindReq	Bind or unbind request: TRUE: bind FALSE: unbind
**psZdpBindReq*	Pointer to request (see above)

Returns

ZPS_E_SUCCESS (request successfully sent)

APS return codes, listed and described in Section 10.2.2

NWK return codes, listed and described in Section 10.2.3

MAC return codes, listed and described in Section 10.2.4

ZPS_eAplZdpBindRegisterRequest

ZPS_teStatus ZPS_eAplZdpBindRegisterRequest(
PDUM_thAPduInstance *hAPduInst*,
ZPS_tuAddress *uDstAddr*,
bool *bExtAddr*,
uint8 **pu8SeqNumber*,
ZPS_tsAplZdpBindRegisterReq **psZdpBindRegisterReq*);

Description

This function informs a remote node with a primary binding table cache that the local node will hold its own binding table entries (and therefore the remote node does not need to hold these entries). The function sends a Bind_Register_req request to the remote node.

The IEEE address of the local node must be specified in the request, which is represented by the structure below (further detailed in Section 8.2.2.25).

```
typedef struct {
    uint64 u64NodeAddress;
} ZPS_tsAplZdpBindRegisterReq;
```

The remote node will reply with a Bind_Register_rsp response, which should be collected using the function **ZQ_bZQueueReceive()** and stored in a structure of type `ZPS_tsAplZdpBindRegisterRsp` (detailed in Section 8.2.3.25). This response contains any information stored about the binding on the remote.

Parameters

hAPduInst	Handle of APDU instance in which request will be sent
uDstAddr	Address of destination node of request (can be 16- or 64-bit, as specified by *bExtAddr*)
bExtAddr	Type of destination address: TRUE: 64-bit IEEE (MAC) address FALSE: 16-bit network address
**pu8SeqNumber*	Pointer to sequence number of request
**psZdpPowerDescReq*	Pointer to request (see above)

Returns

ZPS_E_SUCCESS (request successfully sent)

APS return codes, listed and described in Section 10.2.2

NWK return codes, listed and described in Section 10.2.3

MAC return codes, listed and described in Section 10.2.4

ZPS_eAplZdpReplaceDeviceRequest

```
ZPS_teStatus ZPS_eAplZdpReplaceDeviceRequest(
  PDUM_thAPduInstance hAPduInst,
  ZPS_tuAddress uDstAddr,
  bool bExtAddr,
  uint8 *pu8SeqNumber,
  ZPS_tsAplZdpReplaceDeviceReq *psZdpReplaceDeviceReq);
```

Description

This function requests a remote node with a primary binding table cache to modify binding table entries with new data - more specifically, binding table entries can be modified by replacing an IEEE address and/or associated endpoint number. This function could typically be used in a commissioning application to modify bindings between nodes. The function sends a Replace_Device_req request to the remote node.

This request must include the old IEEE address and its replacement, as well as the corresponding endpoint number and its replacement (if any). The request is represented by the structure below (further detailed in Section 8.2.2.26).

```
typedef struct {
   uint64 u64OldAddress;
   uint8  u8OldEndPoint;
   uint64 u64NewAddress;
   uint8  u8NewEndPoint;
} ZPS_tsAplZdpReplaceDeviceReq;
```

On receiving this request, the remote node will search its binding table for entries containing the old IEEE address and old endpoint number from the request - this pair of values may make up the source or destination data of the binding table entry. These values will be replaced by the new IEEE address and endpoint number from the request. Note that if the endpoint number in the request is zero, only the address will be included in the 'search and replace' (the endpoint number in the modified binding table entries will be left unchanged).

The remote node will check whether a node affected by a binding table change holds a table of its own source bindings (see **ZPS_eAplZdpBindRegisterRequest()**) and, if so, automatically requests an update of this table. The remote node will also request an update of the back-up of the primary binding table cache, if one exists.

The remote node will reply with a Replace_Device_rsp response, which should be collected using the function **ZQ_bZQueueReceive()** and stored in a structure of type `ZPS_tsAplZdpReplaceDeviceRsp` (detailed in Section 8.2.3.26).

Parameters

hAPduInst	Handle of APDU instance in which request will be sent
uDstAddr	Address of destination node of request (can be 16- or 64-bit, as specified by *bExtAddr*)
bExtAddr	Type of destination address: TRUE: 64-bit IEEE (MAC) address FALSE: 16-bit network address
***pu8SeqNumber*	Pointer to sequence number of request
***psZdpReplaceDeviceReq*	Pointer to request (see above)

Returns

ZPS_E_SUCCESS (request successfully sent)

APS return codes, listed and described in Section 10.2.2

NWK return codes, listed and described in Section 10.2.3

MAC return codes, listed and described in Section 10.2.4

ZPS_eAplZdpStoreBkupBindEntryRequest

ZPS_teStatus ZPS_eAplZdpStoreBkupBindEntryRequest(
PDUM_thAPdu *hAPdu*,
ZPS_tuAddress *uDstAddr*,
bool *bExtAddr*,
uint8 **pu8SeqNumber*,
uint16 *u16ProfileId*,
ZPS_tsAplZdpStoreBkupBindEntryReq
psZdpStoreBkupBindEntryReq*);**

Description

This function requests that a back-up of an entry in the local primary binding table cache is performed on a remote node. The destination node of the request must hold the corresponding back-up binding table cache. The back-up operation is normally required when a new entry has been added to the primary binding table cache.

> **Note:** This function is provided in the NXP ZDP API for the reason of interoperability with nodes running non-NXP ZigBee PRO stacks that support the generated request. On receiving a request from this function, the NXP ZigBee PRO stack will return the status ZPS_ZDP_NOT_SUPPORTED.

This request must include the binding table entry to be backed up. The request is represented by the structure below (further detailed in Section 8.2.2.27).

```
typedef struct {
   uint64 u64SrcAddress;
   uint8  u8SrcEndPoint;
   uint16 u16ClusterId;
   uint8  u8DstAddrMode;
   union {
      struct {
         uint16 u16DstAddress;
      } sShort;
      struct {
         uint64 u64DstAddress;
         uint8  u8DstEndPoint;
      } sExtended;
   };
} ZPS_tsAplZdpStoreBkupBindEntryReq;
```

On receiving the request, the remote node adds the specified binding table entry to its back-up binding table cache, if possible.

The remote node replies with a Store_Bkup_Bind_Entry_rsp response, which should be collected using the function **ZQ_bZQueueReceive()** and stored in a structure of type `ZPS_tsAplZdpStoreBkupBindEntryRsp` (detailed in Section 8.2.3.27).

Parameters

hAPdu	Handle of APDU in which request will be sent
uDstAddr	Address of destination node of request (can be 16- or 64-bit, as specified by *bExtAddr*)
bExtAddr	Type of destination address: TRUE: 64-bit IEEE (MAC) address FALSE: 16-bit network address
******pu8SeqNumber*	Pointer to sequence number of request
u16ProfileId	Application profile ID
******psZdpStoreBkupBindEntryReq*	Pointer to request (see above)

Returns

ZPS_E_SUCCESS (request successfully sent)

APS return codes, listed and described in Section 10.2.2

NWK return codes, listed and described in Section 10.2.3

MAC return codes, listed and described in Section 10.2.4

ZPS_eAplZdpRemoveBkupBindEntryRequest

ZPS_teStatus ZPS_eAplZdpRemoveBkupBindEntryRequest(
PDUM_thAPduInstance *hAPduInst*,
ZPS_tuAddress *uDstAddr*,
bool *bExtAddr*,
uint8 **pu8SeqNumber*,
ZPS_tsAplZdpRemoveBkupBindEntryReq
**psZdpRemoveBkupBindEntryReq*);

Description

This function requests the removal of an entry in the back-up binding table cache on a remote node. The function must be called from the node with the corresponding primary binding table cache. The removal of a back-up entry is normally required when an entry in the primary binding table cache has been removed.

Note: This function is provided in the NXP ZDP API for the reason of interoperability with nodes running non-NXP ZigBee PRO stacks that support the generated request. On receiving a request from this function, the NXP ZigBee PRO stack will return the status ZPS_ZDP_NOT_SUPPORTED.

This request must include the binding table entry to be removed. The request is represented by the structure below (further detailed in Section 8.2.2.28).

```
typedef struct {
   uint64 u64SrcAddress;
   uint8  u8SrcEndPoint;
   uint16 u16ClusterId;
   uint8  u8DstAddrMode;
   union {
      struct {
         uint16 u16DstAddress;
      } sShort;
      struct {
         uint64 u64DstAddress;
         uint8  u8DstEndPoint;
      } sExtended;
   };
} ZPS_tsAplZdpRemoveBkupBindEntryReq;
```

On receiving the request, the remote node removes the specified binding table entry from its back-up binding table cache, if possible.

The remote node replies with a Remove_Bkup_Bind_Entry_rsp response, which should be collected using the function **ZQ_bZQueueReceive()** and stored in a structure of type `ZPS_tsAplZdpRemoveBkupBindEntryRsp` (detailed in Section 8.2.3.28).

Parameters

hAPduInst	Handle of APDU instance in which request will be sent
uDstAddr	Address of destination node of request (can be 16- or 64-bit, as specified by *bExtAddr*)
bExtAddr	Type of destination address: TRUE: 64-bit IEEE (MAC) address FALSE: 16-bit network address
***pu8SeqNumber*	Pointer to sequence number of request
***psZdpRemoveBkupBindEntryReq*	Pointer to request (see above)

Returns

ZPS_E_SUCCESS (request successfully sent)

APS return codes, listed and described in Section 10.2.2

NWK return codes, listed and described in Section 10.2.3

MAC return codes, listed and described in Section 10.2.4

ZPS_eAplZdpBackupBindTableRequest

ZPS_teStatus ZPS_eAplZdpBackupBindTableRequest(
PDUM_thAPduInstance *hAPduInst*,
ZPS_tuAddress *uDstAddr*,
bool *bExtAddr*,
uint8 **pu8SeqNumber*,
ZPS_tsAplZdpBackupBindTableReq
******psZdpBackupBindTableReq***);**

Description

This function requests that a back-up of the locally held primary binding table cache is performed on a remote node - the whole or part of the table can be backed up. The destination node of the request must hold the corresponding back-up binding table cache. The latter must already exist and be associated with the cache on the local node through a previous discovery.

Note: This function is provided in the NXP ZDP API for the reason of interoperability with nodes running non-NXP ZigBee PRO stacks that support the generated request. On receiving a request from this function, the NXP ZigBee PRO stack will return the status ZPS_ZDP_NOT_SUPPORTED.

This request must include the binding table entries to be backed up. The request is represented by the structure below (further detailed in Section 8.2.2.29).

```
typedef struct {
   uint16 u16BindingTableEntries;
   uint16 u16StartIndex;
   uint16 u16BindingTableListCount;
   /* Rest of message is variable length */
   ZPS_tsAplZdpBindingTable sBindingTable;
} ZPS_tsAplZdpBackupBindTableReq;
```

On receiving the request, the remote node saves the new binding table, if possible, overwriting existing entries. If the new table is longer than the previous one, as many extra entries as possible will be saved.

The remote node replies with a Backup_Bind_Table_rsp response, which should be collected using the function **ZQ_bZQueueReceive()** and stored in a structure of type `ZPS_tsAplZdpBackupBindTableRsp` (detailed in Section 8.2.3.29).

Parameters

hAPduInst	Handle of APDU instance in which request will be sent
uDstAddr	Address of destination node of request (can be 16- or 64-bit, as specified by *bExtAddr*)
bExtAddr	Type of destination address: TRUE: 64-bit IEEE (MAC) address FALSE: 16-bit network address
***pu8SeqNumber*	Pointer to sequence number of request
***psZdpBackupBindTableReq*	Pointer to request (see above)

Returns

ZPS_E_SUCCESS (request successfully sent)

APS return codes, listed and described in Section 10.2.2

NWK return codes, listed and described in Section 10.2.3

MAC return codes, listed and described in Section 10.2.4

ZPS_eAplZdpRecoverBindTableRequest

```
ZPS_teStatus ZPS_eAplZdpRecoverBindTableRequest(
  PDUM_thAPduInstance hAPduInst,
  ZPS_tuAddress uDstAddr,
  bool bExtAddr,
  uint8 *pu8SeqNumber,
  ZPS_tsAplZdpRecoverBindTableReq
                    *psZdpRecoverBindTableReq);
```

Description

This function requests that a back-up of the locally held primary binding table cache is recovered from a remote node. The destination node of the request must hold the back-up binding table cache which is associated with the primary cache on the local node.

Note: This function is provided in the NXP ZDP API for the reason of interoperability with nodes running non-NXP ZigBee PRO stacks that support the generated request. On receiving a request from this function, the NXP ZigBee PRO stack will return the status ZPS_ZDP_NOT_SUPPORTED.

This request must indicate the starting index in the binding table for the recovery. The request is represented by the structure below (further detailed in Section 8.2.2.30).

```
typedef struct {
   uint16 u16StartIndex;
} ZPS_tsAplZdpRecoverBindTableReq;
```

The remote node replies with a Recover_Bind_Table_rsp response containing the required binding table entries, which should be collected using the function **ZQ_bZQueueReceive()** and stored in a structure of type `ZPS_tsAplZdpRecoverBindTableRsp` (detailed in Section 8.2.3.30). As many binding entries as possible are included in this response. If the returned binding table is incomplete, this is indicated in the response and this function must be called again, with the appropriate starting index, to recover the rest of the table.

Parameters

hAPduInst	Handle of APDU instance in which request will be sent
uDstAddr	Address of destination node of request (can be 16- or 64-bit, as specified by *bExtAddr*)
bExtAddr	Type of destination address: TRUE: 64-bit IEEE (MAC) address FALSE: 16-bit network address
**pu8SeqNumber*	Pointer to sequence number of request
**psZdpRecoverBindTableReq*	Pointer to request (see above)

Returns

ZPS_E_SUCCESS (request successfully sent)

APS return codes, listed and described in Section 10.2.2

NWK return codes, listed and described in Section 10.2.3

MAC return codes, listed and described in Section 10.2.4

ZPS_eAplZdpBackupSourceBindRequest

ZPS_teStatus ZPS_eAplZdpBackupSourceBindRequest(
PDUM_thAPduInstance *hAPduInst*,
ZPS_tuAddress *uDstAddr*,
bool *bExtAddr*,
uint8 **pu8SeqNumber*,
ZPS_tsAplZdpBackupSourceBindReq
psZdpBackupSourceBindReq*);**

Description

This function requests that a back-up of the locally held source binding table is performed on a remote node. This source binding table contains entries only relevant to the local node. The function must be called from a node with a primary binding table cache and the destination node of the request must hold the corresponding back-up binding table cache.

Note: This function is provided in the NXP ZDP API for the reason of interoperability with nodes running non-NXP ZigBee PRO stacks that support the generated request. On receiving a request from this function, the NXP ZigBee PRO stack will return the status ZPS_ZDP_NOT_SUPPORTED.

This request must include the source binding table entries to be backed up. The request is represented by the structure below (further detailed in Section 8.2.2.31).

```
typedef struct {
   uint16 u16SourceTableEntries;
   uint16 u16StartIndex;
   uint16 u16SourceTableListCount;
   /* Rest of message is variable length */
   uint64* pu64SourceAddress;
} ZPS_tsAplZdpBackupSourceBindReq;
```

On receiving the request, the remote node saves the new source binding table, if possible, overwriting existing entries. If the new table is longer than the previous one, as many extra entries as possible will be saved.

The remote node replies with a Backup_Source_Bind_rsp response, which should be collected using the function **ZQ_bZQueueReceive()** and stored in a structure of type `ZPS_tsAplZdpBackupSourceBindRsp` (detailed in Section 8.2.3.31).

Parameters

hAPduInst	Handle of APDU instance in which request will be sent
uDstAddr	Address of destination node of request (can be 16- or 64-bit, as specified by *bExtAddr*)
bExtAddr	Type of destination address: TRUE: 64-bit IEEE (MAC) address FALSE: 16-bit network address
**pu8SeqNumber*	Pointer to sequence number of request
**psZdpBackupSourceBindReq*	Pointer to request (see above)

Returns

ZPS_E_SUCCESS (request successfully sent)

APS return codes, listed and described in Section 10.2.2

NWK return codes, listed and described in Section 10.2.3

MAC return codes, listed and described in Section 10.2.4

ZPS_eAplZdpRecoverSourceBindRequest

```
ZPS_teStatus ZPS_eAplZdpRecoverSourceBindRequest(
  PDUM_thAPduInstance hAPduInst,
  ZPS_tuAddress uDstAddr,
  bool bExtAddr,
  uint8 *pu8SeqNumber,
  ZPS_tsAplZdpRecoverSourceBindReq
                    *psZdpRecoverSourceBindReq);
```

Description

This function requests that a back-up of the locally held source binding table is recovered from a remote node. The function must be called from a node with a primary binding table cache and the destination node of the request must hold the corresponding back-up binding table cache.

Note: This function is provided in the NXP ZDP API for the reason of interoperability with nodes running non-NXP ZigBee PRO stacks that support the generated request. On receiving a request from this function, the NXP ZigBee PRO stack will return the status ZPS_ZDP_NOT_SUPPORTED.

This request must indicate the starting index in the binding table for the recovery. The request is represented by the structure below (further detailed in Section 8.2.2.32).

```
typedef struct {
   uint16 u16StartIndex;
} ZPS_tsAplZdpRecoverSourceBindReq;
```

The remote node replies with a Recover_Source_Bind_rsp response containing the required binding table entries, which should be collected using the function **ZQ_bZQueueReceive()** and stored in a structure of type `ZPS_tsAplZdpRecoverSourceBindRsp` (detailed in Section 8.2.3.32). As many binding entries as possible are included in this response. If the returned binding table is incomplete, this is indicated in the response and this function must be called again, with the appropriate starting index, to recover the rest of the table.

Parameters

hAPduInst	Handle of APDU instance in which request will be sent
uDstAddr	Address of destination node of request (can be 16- or 64-bit, as specified by *bExtAddr*)
bExtAddr	Type of destination address: TRUE: 64-bit IEEE (MAC) address FALSE: 16-bit network address
**pu8SeqNumber*	Pointer to sequence number of request
**psZdpRecoverSourceBindReq*	Pointer to request (see above)

Returns

ZPS_E_SUCCESS (request successfully sent)

APS return codes, listed and described in Section 10.2.2

NWK return codes, listed and described in Section 10.2.3

MAC return codes, listed and described in Section 10.2.4

ZPS_eAplAibRemoveBindTableEntryForMacAddress

ZPS_teStatus
ZPS_eAplAibRemoveBindTableEntryForMacAddress(
uint64 *u64MacAddress***);**

Description

This function requests the removal of the entry corresponding to the specified IEEE/MAC address from the local binding table.

Parameters

u64MacAddress	IEEE/MAC address contained in the binding table entry to be removed

Returns

ZPS_E_SUCCESS

8.1.4 Network Management Services Functions

The ZDP Network Management Services functions are concerned with requests for network operations to be implemented remotely.

The functions are listed below, along with their page references:

Note: Some of these functions cannot be used to send requests to nodes that run the NXP ZigBee PRO stack. They are supplied in the ZDP API in order to facilitate interoperability with nodes based on non-NXP software which supports the corresponding requests.

ZPS_eAplZdpMgmtNwkDiscRequest

ZPS_teStatus ZPS_eAplZdpMgmtNwkDiscRequest(
PDUM_thAPduInstance *hAPduInst*,
ZPS_tuAddress *uDstAddr*,
bool *bExtAddr*,
uint8 ******pu8SeqNumber*,
ZPS_tsAplZdpMgmtNwkDiscReq
******psZdpMgmtNwkDiscReq***);**

Description

This function requests a remote node to perform a channel scan in order to discover any other wireless networks that are operating in the neighbourhood.

> **Note:** This function is provided in the ZDP API for the reason of interoperability with nodes running non-NXP ZigBee PRO stacks that support the generated request. On receiving a request from this function, the NXP ZigBee PRO stack will return the status ZPS_ZDP_NOT_SUPPORTED.

This request must specify the requirements for the scan: channels to scan, duration of scan, starting channel. The request is represented by the structure below (further detailed in Section 8.2.2.33).

```
typedef struct {
   uint32 u32ScanChannels;
   uint8  u8ScanDuration;
   uint8  u8StartIndex;
} ZPS_tsAplZdpMgmtNwkDiscReq;
```

The remote node replies with a Mgmt_NWK_Disc_rsp response containing the scan results, which should be collected using the function **ZQ_bZQueueReceive()** and stored in a structure of type `ZPS_tsAplZdpMgmtNwkDiscRsp` (detailed in Section 8.2.3.33).

Parameters

hAPduInst	Handle of APDU instance in which request will be sent
uDstAddr	Address of destination node of request (can be 16- or 64-bit, as specified by *bExtAddr*)
bExtAddr	Type of destination address: TRUE: 64-bit IEEE (MAC) address FALSE: 16-bit network address
**pu8SeqNumber*	Pointer to sequence number of request
**psZdpMgmtNwkDiscReq*	Pointer to request (see above)

Returns

ZPS_E_SUCCESS (request successfully sent)

APS return codes, listed and described in Section 10.2.2

NWK return codes, listed and described in Section 10.2.3

MAC return codes, listed and described in Section 10.2.4

ZPS_eAplZdpMgmtLqiRequest

```
ZPS_teStatus ZPS_eAplZdpMgmtLqiRequest(
        PDUM_thAPduInstance hAPduInst,
        ZPS_tuAddress uDstAddr,
        bool bExtAddr,
        uint8 *pu8SeqNumber,
        ZPS_tsAplZdpMgmtLqiReq *psZdpMgmtLqiReq);
```

Description

This function requests a remote node to provide a list of neighbouring nodes, from its Neighbour table, including LQI (link quality) values for radio transmissions from each of these nodes. The destination node of this request must be a Router or the Co-ordinator.

This request must specify the index of the first node in the Neighbour table to report. The request is represented by the structure below (further detailed in Section 8.2.2.34).

```
typedef struct {
   uint8  u8StartIndex;
} ZPS_tsAplZdpMgmtLqiReq;
```

The remote node replies with a Mgmt_Lqi_rsp response containing the required information, which should be collected using the function **ZQ_bZQueueReceive()** and stored in a structure of type `ZPS_tsAplZdpMgmtLqiRsp` (detailed in Section 8.2.3.34).

Parameters

hAPduInst	Handle of APDU instance in which request will be sent
uDstAddr	Address of destination node of request (can be 16- or 64-bit, as specified by *bExtAddr*)
bExtAddr	Type of destination address: TRUE: 64-bit IEEE (MAC) address FALSE: 16-bit network address
**pu8SeqNumber*	Pointer to sequence number of request
**psZdpMgmtLqiReq*	Pointer to request (see above)

Returns

ZPS_E_SUCCESS (request successfully sent)

APS return codes, listed and described in Section 10.2.2

NWK return codes, listed and described in Section 10.2.3

MAC return codes, listed and described in Section 10.2.4

ZPS_eAplZdpMgmtRtgRequest

```
ZPS_teStatus ZPS_eAplZdpMgmtRtgRequest(
        PDUM_thAPduInstance hAPduInst,
        ZPS_tuAddress uDstAddr,
        bool bExtAddr,
        uint8 *pu8SeqNumber,
        ZPS_tsAplZdpMgmtRtgReq *psZdpMgmtRtgReq);
```

Description

This function requests a remote node to provide the contents of its Routing table. The destination node of this request must be a Router or the Co-ordinator.

This request must specify the index of the first entry in the Routing table to report. The request is represented by the structure below (further detailed in Section 8.2.2.35).

```
typedef struct {
   uint8  u8StartIndex;
} ZPS_tsAplZdpMgmtRtgReq;
```

The remote node replies with a Mgmt_Rtg_rsp response containing the required information, which should be collected using the function **ZQ_bZQueueReceive()** and stored in a structure of type `ZPS_tsAplZdpMgmtRtgRsp` (detailed in Section 8.2.3.35).

Parameters

hAPduInst	Handle of APDU instance in which request will be sent
uDstAddr	Address of destination node of request (can be 16- or 64-bit, as specified by *bExtAddr*)
bExtAddr	Type of destination address: TRUE: 64-bit IEEE (MAC) address FALSE: 16-bit network address
**pu8SeqNumber*	Pointer to sequence number of request
**psZdpMgmtRtgReq*	Pointer to request (see above)

Returns

ZPS_E_SUCCESS (request successfully sent)

APS return codes, listed and described in Section 10.2.2

NWK return codes, listed and described in Section 10.2.3

MAC return codes, listed and described in Section 10.2.4

ZPS_eAplZdpMgmtBindRequest

ZPS_teStatus ZPS_eAplZdpMgmtBindRequest(
PDUM_thAPduInstance *hAPduInst*,
ZPS_tuAddress *uDstAddr*,
bool *bExtAddr*,
uint8 **pu8SeqNumber*,
ZPS_tsAplZdpMgmtBindReq **psZdpMgmtBindReq*);

Description

This function requests a remote node to provide the contents of its Binding table. The destination node of this request must be a Router or the Co-ordinator.

This request must specify the index of the first entry in the Binding table to report. The request is represented by the structure below (further detailed in Section 8.2.2.36).

```
typedef struct {
   uint8  u8StartIndex;
} ZPS_tsAplZdpMgmtBindReq;
```

The remote node replies with a Mgmt_Bind_rsp response containing the required information, which should be collected using the function **ZQ_bZQueueReceive()** and stored in a structure of type `ZPS_tsAplZdpMgmtBindRsp` (detailed in Section 8.2.3.36).

Parameters

hAPduInst	Handle of APDU instance in which request will be sent
uDstAddr	Address of destination node of request (can be 16- or 64-bit, as specified by *bExtAddr*)
bExtAddr	Type of destination address: TRUE: 64-bit IEEE (MAC) address FALSE: 16-bit network address
**pu8SeqNumber*	Pointer to sequence number of request
**psZdpMgmtBindReq*	Pointer to request (see above)

Returns

ZPS_E_SUCCESS (request successfully sent)

APS return codes, listed and described in Section 10.2.2

NWK return codes, listed and described in Section 10.2.3

MAC return codes, listed and described in Section 10.2.4

ZPS_eAplZdpMgmtLeaveRequest

ZPS_teStatus ZPS_eAplZdpMgmtLeaveRequest(
PDUM_thAPduInstance *hAPduInst*,
ZPS_tuAddress *uDstAddr*,
bool *bExtAddr*,
uint8 **pu8SeqNumber*,
ZPS_tsAplZdpMgmtLeaveReq **psZdpMgmtLeaveReq*);

Description

This function requests a remote node to leave the network. The request also indicates whether the children of the leaving node should also be requested to leave and whether the leaving node(s) should subsequently attempt to rejoin the network.

Note: This function is provided in the ZDP API for the reason of interoperability with nodes running non-NXP ZigBee PRO stacks that support the generated request. On receiving a request from this function, the NXP ZigBee PRO stack will return the status ZPS_ZDP_NOT_SUPPORTED.

The IEEE address of the node to leave the network must be included in the request, as well as flags indicating the children and rejoin choices (see above). The request is represented by the structure below (further detailed in Section 8.2.2.37).

```
typedef struct {
   uint64 u64DeviceAddress;
   uint8  u8Flags;
} ZPS_tsAplZdpMgmtLeaveReq;
```

The remote node replies with a Mgmt_Leave_rsp response, which should be collected using the function **ZQ_bZQueueReceive()** and stored in a structure of type `ZPS_tsAplZdpMgmtLeaveRsp` (detailed in Section 8.2.3.37).

Parameters

hAPduInst	Handle of APDU instance in which request will be sent
uDstAddr	Address of destination node of request (can be 16- or 64-bit, as specified by *bExtAddr*)
bExtAddr	Type of destination address: TRUE: 64-bit IEEE (MAC) address FALSE: 16-bit network address
**pu8SeqNumber*	Pointer to sequence number of request
**psZdpMgmtLeaveReq*	Pointer to request (see above)

Returns

ZPS_E_SUCCESS (request successfully sent)

APS return codes, listed and described in Section 10.2.2

NWK return codes, listed and described in Section 10.2.3

MAC return codes, listed and described in Section 10.2.4

ZPS_eAplZdpMgmtDirectJoinRequest

ZPS_teStatus ZPS_eAplZdpMgmtDirectJoinRequest(
PDUM_thAPduInstance *hAPduInst*,
ZPS_tuAddress *uDstAddr*,
bool *bExtAddr*,
uint8 ****pu8SeqNumber*,
ZPS_tsAplZdpMgmtDirectJoinReq
******psZdpMgmtDirectJoinReq*);

Description

This function requests a remote node to allow a particular device (identified through its IEEE address) to join the network as a child of the node. Thus, joining should be enabled on the remote node just for the nominated device. The destination node of this request must be a Router or the Co-ordinator.

> **Note:** This function is provided in the ZDP API for the reason of interoperability with nodes running non-NXP ZigBee PRO stacks that support the generated request. On receiving a request from this function, the NXP ZigBee PRO stack will return the status ZPS_ZDP_NOT_SUPPORTED.

The IEEE address of the nominated device as well as its capabilities must be included in the request. The request is represented by the structure below (further detailed in Section 8.2.2.38).

```
typedef struct {
   uint64 u64DeviceAddress;
   uint8  u8Capability;
} ZPS_tsAplZdpMgmtDirectJoinReq;
```

The remote node replies with a Mgmt_Direct_Join_req response, which should be collected using the function **ZQ_bZQueueReceive()** and stored in a structure of type `ZPS_tsAplZdpMgmtDirectJoinRsp` (detailed in Section 8.2.3.38).

Parameters

hAPduInst	Handle of APDU instance in which request will be sent
uDstAddr	Address of destination node of request (can be 16- or 64-bit, as specified by *bExtAddr*)
bExtAddr	Type of destination address: TRUE: 64-bit IEEE (MAC) address FALSE: 16-bit network address
**pu8SeqNumber*	Pointer to sequence number of request
**psZdpMgmtDirectJoinReq*	Pointer to request (see above)

Returns

ZPS_E_SUCCESS (request successfully sent)

APS return codes, listed and described in Section 10.2.2

NWK return codes, listed and described in Section 10.2.3

MAC return codes, listed and described in Section 10.2.4

ZPS_eAplZdpMgmtPermitJoiningRequest

ZPS_teStatus ZPS_eAplZdpMgmtPermitJoiningRequest(
PDUM_thAPduInstance *hAPduInst*,
ZPS_tuAddress *uDstAddr*,
bool *bExtAddr*,
uint8 ***pu8SeqNumber*,
ZPS_tsAplZdpMgmtPermitJoiningReq
***psZdpMgmtPermitJoiningReq*);**

Description

This function requests a remote node to enable or disable joining for a specified amount of time. The destination node of this request must be a Router or the Co-ordinator. The request can be unicast to a particular node or broadcast to all routing nodes (for which the destination address must be set to the 16-bit network address 0xFFFC).

Note: This function is provided in the ZDP API for the reason of interoperability with nodes running non-NXP ZigBee PRO stacks that support the generated request. On receiving a request from this function, the NXP ZigBee PRO stack will return the status ZPS_ZDP_NOT_SUPPORTED.

The duration of the enable or disable joining state must be specified in the request. The request is represented by the structure below (further detailed in Section 8.2.2.39).

```
typedef struct {
    uint8 u8PermitDuration;
    bool_t bTcSignificance;
} ZPS_tsAplZdpMgmtPermitJoiningReq;
```

If the request was unicast, the remote node replies with a Mgmt_Permit_Joining_rsp response, which should be collected using the function **ZQ_bZQueueReceive()** and stored in a structure of type `ZPS_tsAplZdpMgmtPermitJoiningRsp` (detailed in Section 8.2.3.39).

Parameters

hAPduInst	Handle of APDU instance in which request will be sent
uDstAddr	Address of destination node of request (can be 16- or 64-bit, as specified by *bExtAddr*)
bExtAddr	Type of destination address: TRUE: 64-bit IEEE (MAC) address FALSE: 16-bit network address
**pu8SeqNumber*	Pointer to sequence number of request
**psZdpMgmtPermitJoiningReq*	Pointer to request (see above)

Returns

ZPS_E_SUCCESS (request successfully sent)

APS return codes, listed and described in Section 10.2.2

NWK return codes, listed and described in Section 10.2.3

MAC return codes, listed and described in Section 10.2.4

ZPS_eAplZdpMgmtCacheRequest

ZPS_teStatus ZPS_eAplZdpMgmtCacheRequest(
PDUM_thAPduInstance *hAPduInst*,
ZPS_tuAddress *uDstAddr*,
bool *bExtAddr*,
**uint8 ****pu8SeqNumber*,
**ZPS_tsAplZdpMgmtCacheReq ****psZdpMgmtCacheReq*);

Description

This function requests a remote node to provide a list of the End Devices registered in its primary discovery cache. Therefore, the destination node must contain a primary discovery cache.

Note: This function is provided in the ZDP API for the reason of interoperability with nodes running non-NXP ZigBee PRO stacks that support the generated request. On receiving a request from this function, the NXP ZigBee PRO stack will return the status ZPS_ZDP_NOT_SUPPORTED.

The request is represented by the structure below (further detailed in Section 8.2.2.40).

```
typedef struct {
   uint8 u8StartIndex;
} ZPS_tsAplZdpMgmtCacheReq;
```

The remote node replies with a Mgmt_Cache_rsp response, which should be collected using the function **ZQ_bZQueueReceive()** and stored in a structure of type `ZPS_tsAplZdpMgmtCacheRsp` (detailed in Section 8.2.3.40).

Parameters

hAPduInst	Handle of APDU in which request will be sent
uDstAddr	Address of destination node of request (can be 16- or 64-bit, as specified by *bExtAddr*)
bExtAddr	Type of destination address: TRUE: 64-bit IEEE (MAC) address FALSE: 16-bit network address
**pu8SeqNumber*	Pointer to sequence number of request
**psZdpMgmtCacheReq*	Pointer to request (see above)

Returns

ZPS_E_SUCCESS (request successfully sent)

APS return codes, listed and described in Section 10.2.2

NWK return codes, listed and described in Section 10.2.3

MAC return codes, listed and described in Section 10.2.4

ZPS_eAplZdpMgmtNwkUpdateRequest

ZPS_teStatus ZPS_eAplZdpMgmtNwkUpdateRequest(
PDUM_thAPduInstance *hAPduInst*,
ZPS_tuAddress *uDstAddr*,
bool *bExtAddr*,
uint8 **pu8SeqNumber*,
ZPS_tsAplZdpMgmtNwkUpdateReq
**psZdpMgmtNwkUpdateReq*);

Description

This function requests an update of network parameters related to radio communication. The request can specify any of the following:

- update the radio channel mask (for scans) and the 16-bit network address of the network manager (node nominated to manage radio-band operation of network)
- change the radio channel used
- scan radio channels and report the results

The request can be broadcast or unicast to nodes with radio receivers that are configured to remain on during idle periods.

The request is represented by the structure below (further detailed in Section 8.2.2.41).

```
typedef struct {
   uint32 u32ScanChannels;
   uint8  u8ScanDuration;
   uint8  u8ScanCount;
   uint8  u8NwkUpdateId;
   uint16 u16NwkManagerAddr;
} ZPS_tsAplZdpMgmtNwkUpdateReq;
```

The specific action to be taken as a result of this request is indicated through the element `u8ScanDuration`, as described in the table below.

`u8ScanDuration`	Action
0x00-0x05	Perform radio channel scan on the set of channels specified through `u32ScanChannels`. The time, in seconds, spent scanning each channel is determined by the value of `u8ScanDuration` and the number of scans is equal to the value of `u8ScanCount`. Valid for unicasts only.
0x06-0xFD	Reserved
0xFE	Change radio channel to single channel specified through `u32ScanChannels` and set the network manager address to that specified through `u16NwkManagerAddr`. Valid for broadcasts only.
0xFF	Update the stored radio channel mask with that specified through `u32ScanChannels` (but do not scan). Valid for broadcasts only.

The remote node replies with a Mgmt_NWK_Update_notify notification, which should be collected using the function **ZQ_bZQueueReceive()** and stored in a structure of type `ZPS_tsAplZdpMgmtNwkUpdateNotify` (detailed in Section 8.2.3.41).

Parameters

hAPduInst	Handle of APDU instance in which request will be sent
uDstAddr	Address of destination node of request (can be 16- or 64-bit, as specified by *bExtAddr*)
bExtAddr	Type of destination address: TRUE: 64-bit IEEE (MAC) address FALSE: 16-bit network address
**pu8SeqNumber*	Pointer to sequence number of request
**psZdpMgmtNwkUpdateReq*	Pointer to request (see above)

Returns

ZPS_E_SUCCESS (request successfully sent)

APS return codes, listed and described in Section 10.2.2

NWK return codes, listed and described in Section 10.2.3

MAC return codes, listed and described in Section 10.2.4

ZPS_eAplZdpParentAnnceReq

```
ZPS_teStatus ZPS_eAplZdpParentAnnceReq(
    PDUM_thAPduInstance hAPduInst,
    ZPS_tuAddress uDstAddr,
    bool bExtAddr,
    uint8 *pu8SeqNumber,
    ZPS_tsAplZdpParentAnnceReq *psZdpParentAnnceReq);
```

Description

This function is used on a Router or the Co-ordinator to send a Parent Announcement message to one or more other nodes. In this announcement, the originating node declares which nodes it has as children. These child nodes are specified using their IEEE/MAC addresses.

The message contains the above data in following structure (further detailed in Section 8.2.2.42):

```
typedef struct {
    uint8    u8NumberOfChildren;
    uint64*  pu64ChildList;
} ZPS_tsAplZdpParentAnnceReq;
```

If a node which receives this message also has one of the specified nodes as its child (so there is a conflict), the receiving node broadcasts a response to indicate this. The response data is contained in the structure below (further detailed in Section 8.2.3.42):

```
typedef struct {
    uint64* pu64ChildList;
    uint8 u8NumberOfChildren;
    uint8 u8Status;
} ZPS_tsAplZdpParentAnnceRsp;
```

Parameters

hAPduInst	Handle of APDU instance in which message will be sent
uDstAddr	Address of destination node of message (can be 16- or 64-bit, as specified by *bExtAddr*)
bExtAddr	Type of destination address: TRUE: 64-bit IEEE (MAC) address FALSE: 16-bit network address
**pu8SeqNumber*	Pointer to sequence number of message
**psZdpParentAnnceReq*	Pointer to message (see above)

Returns

ZPS_E_SUCCESS (request successfully sent)

APS return codes, listed and described in Section 10.2.2

NWK return codes, listed and described in Section 10.2.3

MAC return codes, listed and described in Section 10.2.4

8.1.5 Response Data Extraction Function

The ZDP Response Data Extraction function is concerned with obtaining the data from a received response packet which is destined for the ZDO. The function should be called when a ZPS_EVENT_APS_DATA_INDICATION event is generated for destination endpoint 0.

The function is listed below, along with its page reference:

Note: This function and the related structure `ZPS_tsAfZdpEvent` are defined in the header file **appZdpExtraction.h**.

ZPS_bAplZdpUnpackResponse

bool ZPS_bAplZdpUnpackResponse(
ZPS_tsAfEvent **psZdoServerEvent*,
ZPS_tsAfZdpEvent **psReturnStruct*);

Description

This function can be used to extract data received in a response packet which is destined for the ZDO (at endpoint 0). When such a packet is received, the event ZPS_EVENT_APS_DATA_INDICATION is generated. The application must then check whether the destination endpoint number is 0 in the event and, if this is the case, call this function to extract the response data from the event.

A pointer to a `ZPS_tsAfZdpEvent` structure must be provided, which the function will populate with the extracted data.

Parameters

******psZdoServerEvent*	Pointer to structure containing the event (see Section 7.2.2.1)
******psReturnStruct*	Pointer to structure to receive extracted data (see Section 7.2.2.25)

Returns

TRUE if data successfully extracted

FALSE if data not successfully extracted

8.2 ZDP Structures

This section describes the structures used by the ZigBee Device Profile (ZDP) API.

Three sets of structures are presented:

- Structures used to represent the descriptors that reside on a node - see Section 8.2.1
- Structures used to issue requests using the ZDP functions - see Section 8.2.2
- Structures used to receive responses to the ZDP requests - see Section 8.2.3

8.2.1 Descriptor Structures

These structures are used to represent the following descriptors that contain information about the host node:

- Node descriptor
- Node Power descriptor
- Simple descriptor

The structures are listed below, along with their page references.

8.2.1.1 ZPS_tsAplZdpNodeDescriptor

The ZDP Node descriptor structure `ZPS_tsAplZdpNodeDescriptor` is shown below.

```
typedef struct {
    union
    {
        ZPS_tsAplZdpNodeDescBitFields sBitFields;
        uint16 u16Value;
    } uBitUnion;
    uint8 u8MacFlags;
    uint16 u16ManufacturerCode;
    uint8 u8MaxBufferSize;
    uint16 u16MaxRxSize;
    uint16 u16ServerMask;
    uint16 u16MaxTxSize;
    uint8 u8DescriptorCapability;
} ZPS_tsAplZdpNodeDescriptor;
```

where:

- `sBitFields` is a structure of the type `ZPS_tsAplZdpNodeDescBitFields` (described below) containing various items of information about the node.
- `u16Value` is used for the union and should be set to 0x0000.
- `eMacFlags` contains 8 bits (bits 0-7) indicating the node capabilities, as required by the IEEE 802.15.4 MAC sub-layer. These node capability flags are described in Table 8 on page 199.
- `u16ManufacturerCode` contains 16 bits (bits 0-15) indicating the manufacturer code for the node, where this code is allocated to the manufacturer by the ZigBee Alliance.
- `u8MaxBufferSize` is the maximum size, in bytes, of an NPDU (Network Protocol Data Unit).
- `u16MaxRxSize` is the maximum size, in bytes, of an APDU (Application Protocol Data Unit). This value can be greater than the value of `u8MaxBufferSize`, due to the fragmentation of an APDU into NPDUs.
- `u16ServerMask` contains 8 bits (bits 0-7) indicating the server status of the node. This server mask is detailed in Table 18 on page 389.
- `u16MaxTxSize` is the maximum size, in bytes, of the ASDU (Application Sub-layer Data Unit) in which a message can be sent (the message may actually be transmitted in smaller fragments)
- `u8DescriptorCapability` contains 8 bits (bits 0-7) indicating the properties of the node that can be used by other nodes in network discovery, as indicated in the table below.

Bit	Description
0	Set to 1 if Extended Active Endpoint List is available on the node, 0 otherwise
1	Set to 1 if Extended Simple Descriptor List is available on the node, 0 otherwise
2-7	Reserved

ZPS_tsAplZdpNodeDescBitFields

The `ZPS_tsAplZdpNodeDescBitFields` structure is used by the `sBitFields` element in the Node descriptor structure (see above), and is shown below:

```
typedef struct {
    unsigned eFrequencyBand    : 5;
    unsigned eApsFlags         : 3;
    unsigned eReserved         : 3; /* reserved */
    unsigned bUserDescAvail    : 1;
    unsigned bComplexDescAvail : 1;
    unsigned eLogicalType      : 3;
}ZPS_tsAplZdpNodeDescBitFields;
```

where:

- `eFrequencyBand` is a 5-bit value representing the IEEE 802.15.4 radio-frequency band used by the node:
  - 0: 868-MHz band
  - 2: 915-MHz band
  - 3: 2400-MHz band
- `eApsFlags` is a 3-bit value containing flags that indicate the ZigBee APS capabilities of the node (not currently supported and should be set to 0).
- `eReserved` is a 3-bit reserved value.
- `bUserDescAvail` is a 1-bit value indicating whether a User descriptor is available for the node - 1 indicates available, 0 indicates unavailable.
- `bComplexDescAvail` is a 1-bit value indicating whether a Complex descriptor is available for the node - 1 indicates available, 0 indicates unavailable.
- `eLogicalType` is a 3-bit value indicating the ZigBee device of the node:
  - 0: Co-ordinator
  - 1: Router
  - 2: End Device

8.2.1.2 ZPS_tsAplZdpNodePowerDescriptor

The ZDP Node Power descriptor structure `ZPS_tsAplZdpNodePowerDescriptor` is shown below.

```
typedef struct {
 union
    {
        ZPS_tsAplZdpPowerDescBitFields sBitFields;
        uint16 u16Value;
    }uBitUnion;
} ZPS_tsAplZdpNodePowerDescriptor;
```

where:

- `sBitFields` is a structure of type `ZPS_tsAplZdpPowerDescBitFields` (described below) containing various items of information about the node's power.
- `u16value` is used for the union and should be set to 0x0000.

ZPS_tsAplZdpPowerDescBitFields

The `ZPS_tsAplZdpPowerDescBitFields` structure is used by the `sBitFields` element in the Node Power descriptor structure (see above), and is shown below:

```
typedef struct {
   unsigned eCurrentPowerSourceLevel   : 4;
   unsigned eCurrentPowerSource        : 4;
   unsigned eAvailablePowerSource      : 4;
   unsigned eCurrentPowerMode          : 4;
}ZPS_tsAplZdpPowerDescBitFields;
```

where:

- `eCurrentPowerSourceLevel` is a 4-bit value roughly indicating the level of charge of the node's power source (mainly useful for batteries), as follows:
  - 0000: Critically low
  - 0100: Approximately 33%
  - 1000: Approximately 66%
  - 1100: Approximately 100% (near fully charged)
- `eCurrentPowerSource` is a 4-bit value indicating the current power source for the node, as detailed below (the bit corresponding to the current power source is set to 1, all other bits are set to 0):
  - Bit 0: Permanent mains supply
  - Bit 1: Rechargeable battery
  - Bit 2: Disposable battery
  - Bit 4: Reserved
- `eAvailablePowerSource` is a 4-bit value indicating the available power sources for the node, as detailed above (a bit is set to 1 if the corresponding power source is available).
- `eCurrentPowerMode` is a 4-bit value indicating the power mode currently used by the node, as follows:
  - 0000: Receiver synchronised with the "receiver on when idle" subfield of the Node descriptor
  - 0001: Receiver switched on periodically, as defined by the Node Power descriptor
  - 0010: Receiver switched on when stimulated, e.g. by pressing a button
  - All other values are reserved

8.2.1.3 ZPS_tsAplZdpSimpleDescType

The ZDP Simple descriptor structure ZPS_tsAplZdpSimpleDescType is shown below.

```
typedef struct {
    uint8  u8Endpoint;
    uint16 u16ApplicationProfileId;
    uint16 u16DeviceId;
    union
    {
        ZPS_tsAplZdpSimpleDescBitFields sBitFields;
        uint8 u8Value;
    }uBitUnion;
    uint8  u8InClusterCount;
    uint16* pu16InClusterList;
    uint8  u8OutClusterCount;
    uint16* pu16OutClusterList;
}ZPS_tsAplZdpSimpleDescType;
```

where:

- u8Endpoint is the number, in the range 1-240, of the endpoint to which the Simple descriptor corresponds.
- u16ApplicationProfileId is the 16-bit identifier of the ZigBee application profile supported by the endpoint. This must be an application profile identifier issued by the ZigBee Alliance (for Lighting & Occupancy devices, it is 0x0104).
- u16DeviceId is the 16-bit identifier of the ZigBee device description supported by the endpoint. This must be a device description identifier issued by the ZigBee Alliance.
- sBitFields is a structure of type ZPS_tsAplZdpSimpleDescBitFields (described below) containing information about the endpoint.
- u8Value is used for the union and must be set to 0x00.
- u8InClusterCount is an 8-bit count of the number of input clusters, supported on the endpoint, that will appear in the list pointed to by the pu16InClusterList element.
- *pu16InClusterList is a pointer to the list of input clusters supported by the endpoint (for use during the service discovery and binding procedures). This is a sequence of 16-bit values, representing the cluster numbers (in the range 1-240), where the number of values is equal to count u8InClusterCount. If this count is zero, the pointer can be set to NULL.
- u8OutClusterCount is an 8-bit count of the number of output clusters, supported on the endpoint, that will appear in the pu16OutClusterList element.
- *pu16OutClusterList is a pointer to the list of output clusters supported by the endpoint (for use during the service discovery and binding procedures). This is a sequence of 16-bit values, representing the cluster numbers (in the range 1-240), where the number of values is equal to count u8OutClusterCount. If this count is zero, the pointer can be set to NULL.

ZPS_tsAplZdpSimpleDescBitFields

The `ZPS_tsAplZdpSimpleDescBitFields` structure is used by the `sBitFields` element in the Simple descriptor structure (see above), and is shown below:

```
typedef struct
{
    unsigned eDeviceVersion :4;
    unsigned eReserved      :4;
}ZPS_tsAplZdpSimpleDescBitFields;
```

where:

- `eDeviceVersion` is a 4-bit value identifying the version of the device description supported by the endpoint.
- `eReserved` is a 4-bit reserved value.

8.2.2 ZDP Request Structures

These structures are used to represent requests in the ZDP functions.

The ZDP request structures are listed below, along with their page references.

Network Management Services Request Structures

8.2.2.1 ZPS_tsAplZdpNwkAddrReq

This structure is used by the function **ZPS_eAplZdpNwkAddrRequest()**. It represents a request for the network address of the node with a given IEEE address.

The `ZPS_tsAplZdpNwkAddrReq` structure is detailed below.

```
typedef struct {
   uint64 u64IeeeAddr;
   uint8  u8RequestType;
   uint8  u8StartIndex;
} ZPS_tsAplZdpNwkAddrReq;
```

where:

- `u64IeeeAddr` is the IEEE address of the node of interest
- `u8RequestType` is the type of response required:
  - 0x00: Single device response, which will contain only the network address of the target node
  - 0x01: Extended response, which will also include the network addresses of neighbouring nodes
  - All other values are reserved
- `u8StartIndex` is the Neighbour table index of the first neighbouring node to be included in the response, if an extended response has been selected

8.2.2.2 ZPS_tsAplZdpIEEEAddrReq

This structure is used by the function **ZPS_eAplZdpIEEEAddrRequest()**. It represents a request for the IEEE address of a node with a given network address.

The `ZPS_tsAplZdpIEEEAddrReq` structure is detailed below.

```
typedef struct {
    uint16 u16NwkAddrOfInterest;
    uint8  u8RequestType;
    uint8  u8StartIndex;
} ZPS_tsAplZdpIEEEAddrReq;
```

where:

- `u16NwkAddrOfInterest` is the network address of the node of interest
- `u8RequestType` is the type of response required:
  - 0x00: Single device response, which will contain only the IEEE address of the target node
  - 0x01: Extended response, which will also include the IEEE addresses of neighbouring nodes
  - All other values are reserved
- `u8StartIndex` is the Neighbour table index of the first neighbouring node to be included in the response, if an extended response has been selected

8.2.2.3 ZPS_tsAplZdpDeviceAnnceReq

This structure is used by the function **ZPS_eAplZdpDeviceAnnceRequest()**. It represents an announcement that the sending node has joined or rejoined the network.

The `ZPS_tsAplZdpDeviceAnnceReq` structure is detailed below.

```
typedef struct {
    uint16 u16NwkAddr;
    uint64 u64IeeeAddr;
    uint8  u8Capability;
} ZPS_tsAplZdpDeviceAnnceReq;
```

where:

- `u16NwkAddr` is the network address of the sending node
- `u64IeeeAddr` is the IEEE address of the sending node
- `u8Capability` is a bitmap representing the capabilities of the sending node. This bitmap is detailed in Table 8 on page 199

8.2.2.4 ZPS_tsAplZdpNodeDescReq

This structure is used by the function **ZPS_eAplZdpNodeDescRequest()**. It represents a request for the Node descriptor of the node with a given network address.

The `ZPS_tsAplZdpNodeDescReq` structure is detailed below.

```
typedef struct {
   uint16 u16NwkAddrOfInterest;
} ZPS_tsAplZdpNodeDescReq;
```

where `u16NwkAddrOfInterest` is the network address of the node of interest.

8.2.2.5 ZPS_tsAplZdpPowerDescReq

This structure is used by the function **ZPS_eAplZdpPowerDescRequest()**. It represents a request for the Power descriptor of the node with a given network address.

The `ZPS_tsAplZdpPowerDescReq` structure is detailed below.

```
typedef struct {
   uint16 u16NwkAddrOfInterest;
} ZPS_tsAplZdpPowerDescReq;
```

where `u16NwkAddrOfInterest` is the network address of the node of interest.

8.2.2.6 ZPS_tsAplZdpSimpleDescReq

This structure is used by the function **ZPS_eAplZdpSimpleDescRequest()**. It represents a request for the Simple descriptor of an endpoint on the node with a given network address.

The `ZPS_tsAplZdpSimpleDescReq` structure is detailed below.

```
typedef struct {
   uint16 u16NwkAddrOfInterest;
   uint8  u8EndPoint;
} ZPS_tsAplZdpSimpleDescReq;
```

where:

- `u16NwkAddrOfInterest` is the network address of the node of interest
- `u8EndPoint` is the number of the relevant endpoint on the node (1-240)

8.2.2.7 ZPS_tsAplZdpExtendedSimpleDescReq

This structure is used by the **ZPS_eAplZdpExtendedSimpleDescRequest()** function. It represents a request for the Simple descriptor of an endpoint on the node with a given network address. This request is required when the endpoint has more input/output clusters than the usual **ZPS_eAplZdpSimpleDescRequest()** function can deal with.

The `ZPS_tsAplZdpExtendedSimpleDescReq` structure is detailed below.

```
typedef struct {
   uint16 u16NwkAddr;
   uint8  u8EndPoint;
   uint8  u8StartIndex;
} ZPS_tsAplZdpExtendedSimpleDescReq;
```

where:

- `u16NwkAddrOfInterest` is the network address of the node of interest
- `u8EndPoint` is the number of the relevant endpoint on the node (1-240)
- `u8StartIndex` is the index of the first cluster of interest in the input and output cluster lists for the endpoint (this and subsequent clusters will be reported in the response)

8.2.2.8 ZPS_tsAplZdpComplexDescReq

This structure is used by the function **ZPS_eAplZdpComplexDescRequest()**. It represents a request for the Complex descriptor of the node with a given network address.

The `ZPS_tsAplZdpComplexDescReq` structure is detailed below.

```
typedef struct {
   uint16 u16NwkAddrOfInterest;
} ZPS_tsAplZdpComplexDescReq;
```

where `u16NwkAddrOfInterest` is the network address of the node of interest.

8.2.2.9 ZPS_tsAplZdpUserDescReq

This structure is used by the function **ZPS_eAplZdpUserDescRequest()**. It represents a request for the User descriptor of the node with a given network address.

The `ZPS_tsAplZdpUserDescReq` structure is detailed below.

```
typedef struct {
   uint16 u16NwkAddrOfInterest;
} ZPS_tsAplZdpUserDescReq;
```

where `u16NwkAddrOfInterest` is the network address of the node of interest.

8.2.2.10 ZPS_tsAplZdpMatchDescReq

This structure is used by the function **ZPS_eAplZdpMatchDescRequest()**. It represents a request for nodes with endpoints that match certain criteria in their Simple descriptors.

The `ZPS_tsAplZdpMatchDescReq` structure is detailed below.

```
typedef struct {
    uint16 u16NwkAddrOfInterest;
    uint16 u16ProfileId;
    /* rest of message is variable length */
    uint8  u8NumInClusters;
    uint16* pu16InClusterList;
    uint8  u8NumOutClusters;
    uint16* pu16OutClusterList;
} ZPS_tsAplZdpMatchDescReq;
```

where:

- `u16NwkAddrOfInterest` is the network address of the node of interest
- `u16ProfileId` is the identifier of the ZigBee application profile used
- `u8NumInClusters` is the number of input clusters to be matched
- `pu16InClusterList` is a pointer to the list of input clusters to be matched - this is a variable-length list of input cluster IDs, two bytes for each cluster
- `u8NumOutClusters` is the number of output clusters to be matched
- `pu16OutClusterList` is a pointer to the list of output clusters to be matched - this is a variable-length list of output cluster IDs, two bytes for each cluster

8.2.2.11 ZPS_tsAplZdpActiveEpReq

This structure is used by the function **ZPS_eAplZdpActiveEpRequest()**. It represents a request for a list of the active endpoints on the node with a given network address.

The `ZPS_tsAplZdpActiveEpReq` structure is detailed below.

```
typedef struct {
    uint16 u16NwkAddrOfInterest;
} ZPS_tsAplZdpActiveEpReq;
```

where `u16NwkAddrOfInterest` is the network address of the node of interest.

8.2.2.12 ZPS_tsAplZdpExtendedActiveEpReq

This structure is used by the function **ZPS_eAplZdpExtendedActiveEpRequest()**. It represents a request for a list of the active endpoints on the node with a given network address. This request is required when the node has more active endpoints than the usual **ZPS_eAplZdpActiveEpRequest()** function can deal with.

The `ZPS_tsAplZdpExtendedActiveEpReq` structure is detailed below.

```
typedef struct {
   uint16 u16NwkAddr;
   uint8  u8StartIndex;
} ZPS_tsAplZdpExtendedActiveEpReq;
```

where:

- `u16NwkAddr` is the network address of the node of interest
- `u8StartIndex` is the index of the first endpoint of interest in the list of active endpoints (this and subsequent endpoints will be reported in the response)

8.2.2.13 ZPS_tsAplZdpUserDescSet

This structure is used by the function **ZPS_eAplZdpUserDescSetRequest()**. It represents a request used to configure the User descriptor on a remote node.

The `ZPS_tsAplZdpUserDescSet` structure is detailed below.

```
typedef struct {
    uint16 u16NwkAddrOfInterest;
    uint8  u8Length;
    char szUserDescriptor[ZPS_ZDP_LENGTH_OF_USER_DESC];
} ZPS_tsAplZdpUserDescSet;
```

where:

- `u16NwkAddrOfInterest` is the network address of the node of interest
- `u8Length` is the length of the User descriptor
- `szUserDescriptor` is the new User descriptor for the remote node as a character array.

8.2.2.14 ZPS_tsAplZdpSystemServerDiscoveryReq

This structure is used by the **ZPS_eAplZdpSystemServerDiscoveryRequest()** function. It represents a request for information on the available services of a remote node.

The `ZPS_tsAplZdpSystemServerDiscoveryReq` structure is detailed below.

```
typedef struct {
   uint16 u16ServerMask;
} ZPS_tsAplZdpSystemServerDiscoveryReq;
```

where `u16ServerMask` is a bitmask representing the required services (1 for 'required', 0 for 'not required'). This bitmask is detailed in the table below.

Bit	Service
0	Primary Trust Centre
1	Backup Trust Centre
2	Primary Binding Table Cache
3	Backup Binding Table Cache
4	Primary Discovery Cache
5	Back-up Discovery Cache
6	Network Manager
7-15	Reserved

Table 13: Services Bitmask

8.2.2.15 ZPS_tsAplZdpDiscoveryCacheReq

This structure is used by the function **ZPS_eAplZdpDiscoveryCacheRequest()**. It represents a request to find the nodes in the network which have a primary discovery cache.

The `ZPS_tsAplZdpDiscoveryCacheReq` structure is detailed below.

```
typedef struct {
   uint16 u16NwkAddr;
   uint64 u64IeeeAddr;
} ZPS_tsAplZdpDiscoveryCacheReq;
```

where:

- `u16NwkAddr` is the network address of the sending node
- `u64IeeeAddr` is the IEEE address of the sending node

8.2.2.16 ZPS_tsAplZdpDiscoveryStoreReq

This structure is used by the function **ZPS_eAplZdpDiscoveryStoreRequest()**. It represents a request to a remote node to reserve memory space to store the local node's 'discovery information'.

The `ZPS_tsAplZdpDiscoveryStoreReq` structure is detailed below.

```
typedef struct {
   uint16 u16NwkAddr;
   uint64 u64IeeeAddr;
   uint8  u8NodeDescSize;
   uint8  u8PowerDescSize;
   uint8  u8ActiveEpSize;
   uint8  u8SimpleDescCount;
   /* Rest of message is variable length */
   uint8* pu8SimpleDescSizeList;
} ZPS_tsAplZdpDiscoveryStoreReq;
```

where:

- `u16NwkAddr` is the network address of the sending node
- `u64IeeeAddr` is the IEEE address of the sending node
- `u8NodeDescSize` is the size of the Node descriptor to store
- `u8PowerDescSize` is the size of the Power descriptor to store
- `u8ActiveEpSize` is the size of the list of active endpoints to store
- `u8SimpleDescCount` is the number of Simple descriptors to store
- `pu8SimpleDescSizeList` is a pointer to a list of sizes of the Simple descriptors

8.2.2.17 ZPS_tsAplZdpNodeDescStoreReq

This structure is used by the function **ZPS_eAplZdpNodeDescStoreRequest()**. It represents a request to a remote node to store the Node descriptor of the local node.

The `ZPS_tsAplZdpNodeDescStoreReq` structure is detailed below.

```
typedef struct {
   uint16 u16NwkAddr;
   uint64 u64IeeeAddr;
   /* Rest of message is variable length */
   ZPS_tsAplZdpNodeDescriptor sNodeDescriptor;
} ZPS_tsAplZdpNodeDescStoreReq;
```

where:

- `u16NwkAddr` is the network address of the sending node
- `u64IeeeAddr` is the IEEE address of the sending node
- `sNodeDescriptor` is a pointer to the Node descriptor to store (this is itself a structure of the type `ZPS_tsAplZdpNodeDescriptor`, detailed in Section 8.2.1.1)

8.2.2.18 ZPS_tsAplZdpPowerDescStoreReq

This structure is used by the function **ZPS_eAplZdpPowerDescStoreRequest()**. It represents a request to a remote node to store the Power descriptor of the local node.

The `ZPS_tsAplZdpPowerDescStoreReq` structure is detailed below.

```
typedef struct {
   uint16 u16NwkAddr;
   uint64 u64IeeeAddr;
   /* Rest of message is variable length */
   ZPS_tsAplZdpNodePowerDescriptor sPowerDescriptor;
} ZPS_tsAplZdpPowerDescStoreReq;
```

where:

- `u16NwkAddr` is the network address of the sending node
- `u64IeeeAddr` is the IEEE address of the sending node
- `sPowerDescriptor` is a pointer to the Power descriptor to store (this is itself a structure of the type `ZPS_tsAplZdpNodePowerDescriptor`, detailed in Section 8.2.1.2)

8.2.2.19 ZPS_tsAplZdpSimpleDescStoreReq

This structure is used by the function **ZPS_eAplZdpSimpleDescStoreRequest()**. It represents a request to a remote node to store the Simple descriptor of one of the local node's endpoints.

The `ZPS_tsAplZdpSimpleDescStoreReq` structure is detailed below.

```
typedef struct {
   uint16 u16NwkAddr;
   uint64 u64IeeeAddr;
   uint8  u8Length;
   /* Rest of message is variable length */
   ZPS_tsAplZdpSimpleDescType sSimpleDescriptor;
} ZPS_tsAplZdpSimpleDescStoreReq;
```

where:

- `u16NwkAddr` is the network address of the sending node
- `u64IeeeAddr` is the IEEE address of the sending node
- `u8Length` is the length of the Simple descriptor to store
- `sSimpleDescriptor` is a pointer to the Simple descriptor to store (this is itself a structure of the type `ZPS_tsAplZdpSimpleDescType`, detailed in Section 8.2.1.3)

8.2.2.20 ZPS_tsAplZdpActiveEpStoreReq

This structure is used by the function **ZPS_eAplZdpActiveEpStoreRequest()**. It represents a request to a remote node to store the list of active endpoints of the local node.

The `ZPS_tsAplZdpActiveEpStoreReq` structure is detailed below.

```
typedef struct {
   uint16 u16NwkAddr;
   uint64 u64IeeeAddr;
   uint8  u8ActiveEPCount;
   /* Rest of message is variable length */
   uint8* pu8ActiveEpList;
} ZPS_tsAplZdpActiveEpStoreReq;
```

where:

- `u16NwkAddr` is the network address of the sending node
- `u64IeeeAddr` is the IEEE address of the sending node
- `u8ActiveEPCount` is the number of active endpoints in the list to store
- `pu8ActiveEpList` is a pointer to the list of active endpoints to store

8.2.2.21 ZPS_tsAplZdpFindNodeCacheReq

This structure is used by the function **ZPS_eAplZdpActiveEpStoreRequest()**. It represents a request to search for nodes in the network that hold ‘discovery information’ about a particular node.

The `ZPS_tsAplZdpFindNodeCacheReq` structure is detailed below.

```
typedef struct {
   uint16 u16NwkAddr;
   uint64 u64IeeeAddr;
} ZPS_tsAplZdpFindNodeCacheReq;
```

where:

- `u16NwkAddr` is the network address of the node of interest
- `u64IeeeAddr` is the IEEE address of the node of interest

8.2.2.22 ZPS_tsAplZdpRemoveNodeCacheReq

This structure is used by the function **ZPS_eAplZdpActiveEpStoreRequest()**. It represents a request to a remote node to remove from its Primary Discovery Cache all ‘discovery information’ relating to a particular End Device.

The `ZPS_tsAplZdpRemoveNodeCacheReq` structure is detailed below.

```
typedef struct {
   uint16 u16NwkAddr;
   uint64 u64IeeeAddr;
} ZPS_tsAplZdpRemoveNodeCacheReq;
```

where:

- `u16NwkAddr` is the network address of the End Device of interest
- `u64IeeeAddr` is the IEEE address of the End Device of interest

8.2.2.23 ZPS_tsAplZdpEndDeviceBindReq

This structure is used by the function **ZPS_eAplZdpEndDeviceBindRequest()**. It represents a request to the Co-ordinator to bind an endpoint on the local node to an endpoint on a remote node (the Co-ordinator must match two such binding requests, from the local node and remote node).

The `ZPS_tsAplZdpEndDeviceBindReq` structure is detailed below.

```
typedef struct {
    uint16 u16BindingTarget;
    uint64 u64SrcIeeeAddress;
    uint8  u8SrcEndpoint;
    uint16 u16ProfileId;
    /* Rest of message is variable length */
    uint8 u8NumInClusters;
    uint16 *pu16InClusterList;
    uint8 u8NumOutClusters;
    uint16 *pu16OutClusterList;
} ZPS_tsAplZdpEndDeviceBindReq;
```

where:

- `u16BindingTarget` is the network address of the node to hold the binding (either a node with primary binding table cache or the local node)
- `u64SrcIeeeAddress` is the IEEE address of the local node
- `u8SrcEndpoint` is the number of the local endpoint to be bound (1-240)
- `u16ProfileId` is the application profile ID to be matched for the binding
- `u8NumInClusters` is the number of input clusters of the local endpoint (available for matching with output clusters of remote node to be bound)
- `pu16InClusterList` is a pointer to the input cluster list of the local endpoint (containing clusters for matching with output clusters of remote node)
- `u8NumOutClusters` is the number of output clusters of the local endpoint (available for matching with input clusters of remote node to be bound)
- `pu16OutClusterList` is a pointer to the output cluster list of the local endpoint (containing clusters for matching with input clusters of remote node)

8.2.2.24 ZPS_tsAplZdpBindUnbindReq

This structure is used by the function **ZPS_eAplZdpBindUnbindRequest()**. It represents a request for a modification of the Binding table on the target node, in order to either bind or unbind two nodes in the network.

The `ZPS_tsAplZdpBindUnbindReq` structure is detailed below.

```
typedef struct {
    uint64 u64SrcAddress;
    uint8  u8SrcEndpoint;
    uint16 u16ClusterId;
    uint8  u8DstAddrMode;
    union {
        struct {
            uint16 u16DstAddress;
        } sShort;
        struct {
            uint64 u64DstAddress;
            uint8  u8DstEndPoint;
        } sExtended;
     } uAddressField;
 } ZPS_tsAplZdpBindUnbindReq;
```

where:

- `u64SrcAddress` is the IEEE address of the source node for the binding
- `u8SrcEndpoint` is the number of the source endpoint for the binding (1-240)
- `u16ClusterId` is the ID of the cluster (on the local endpoint) for the binding
- `u8DstAddrMode` is the destination addressing mode (see Table 14 below):
  - ZPS_E_ADDR_MODE_SHORT: network address (`u8DstEndPoint` is unspecified)
  - ZPS_E_ADDR_MODE_IEEE: IEEE address (`u8DstEndPoint` is specified)
  - All other values are reserved
- `u16DstAddress` or `u64DstAddress` is the address of the destination node for the binding:
  - network address `u16DstAddress` if `u8DstAddrMode` is set to ZPS_E_ADDR_MODE_SHORT
  - IEEE address `u64DstAddress` if `8DstAddrMode` is set to ZPS_E_ADDR_MODE_IEEE
- `u8DstEndPoint` is the number of the destination endpoint for the binding (1-240) - not required if `u8DstAddrMode` set to ZPS_E_ADDR_MODE_SHORT (network address)

u8DstAddrMode	Code	Description
0x02	ZPS_E_ADDR_MODE_SHORT	16-bit Network (Short) address
0x03	ZPS_E_ADDR_MODE_IEEE	64-bit IEEE/MAC address

Table 14: Addressing Modes

8.2.2.25 ZPS_tsAplZdpBindRegisterReq

This structure is used by the function **ZPS_eAplZdpBindRegisterRequest()**. It represents a request to inform a remote node with a primary binding table cache that the local node will hold its own Binding table entries.

The `ZPS_tsAplZdpBindRegisterReq` structure is detailed below.

```
typedef struct {
   uint64 u64NodeAddress;
} ZPS_tsAplZdpBindRegisterReq;
```

where `u64NodeAddress` is the IEEE address of the local node.

8.2.2.26 ZPS_tsAplZdpReplaceDeviceReq

This structure is used by the function **ZPS_eAplZdpReplaceDeviceRequest()**. It represents a request to a remote node (with a primary binding table cache) to modify its binding table entries by replacing an IEEE address and/or associated endpoint number.

The `ZPS_tsAplZdpReplaceDeviceReq` structure is detailed below.

```
typedef struct {
   uint64 u64OldAddress;
   uint8  u8OldEndPoint;
   uint64 u64NewAddress;
   uint8  u8NewEndPoint;
} ZPS_tsAplZdpReplaceDeviceReq;
```

where:

- `u64OldAddress` is the IEEE address to be replaced
- `u8OldEndPoint` is the endpoint number to be replaced
  (0-240, where 0 indicates that the endpoint number is not to be replaced)
- `u64NewAddress` is the replacement IEEE address
- `u8NewEndPoint` is the replacement endpoint number (1-240)

8.2.2.27 ZPS_tsAplZdpStoreBkupBindEntryReq

This structure is used by the function **ZPS_eAplZdpStoreBkupBindEntryRequest()**. It represents a request to a remote node to save a back-up of an entry from the local primary binding table cache.

The `ZPS_tsAplZdpStoreBkupBindEntryReq` structure is detailed below.

```
typedef struct {
   uint64 u64SrcAddress;
   uint8  u8SrcEndPoint;
   uint16 u16ClusterId;
   uint8  u8DstAddrMode;
   union {
      struct {
         uint16 u16DstAddress;
      } sShort;
      struct {
         uint64 u64DstAddress;
         uint8  u8DstEndPoint;
      } sExtended;
   };
} ZPS_tsAplZdpStoreBkupBindEntryReq;
```

where:

- `u64SrcAddress` is the IEEE address of the source node for the binding entry
- `u8SrcEndpoint` is the number of the source endpoint for the binding (1-240)
- `u16ClusterId` is the ID of the cluster (on the local endpoint) for the binding
- `u8DstAddrMode` is the destination addressing mode for remaining elements (see Table 15 below)
- `u16DstAddress` is the address of the destination node for the binding (address type according to setting of `u8DstAddrMode`)
- `u8DstEndPoint` is the number of the destination endpoint for the binding (1-240)

u8DstAddrMode	Code	Description
0x01	ZPS_E_ADDR_MODE_GROUP	16-bit Group address
0x03	ZPS_E_ADDR_MODE_IEEE	64-bit IEEE/MAC address

Table 15: Addressing Modes

8.2.2.28 ZPS_tsAplZdpRemoveBkupBindEntryReq

This structure is used by the **ZPS_eAplZdpRemoveBkupBindEntryRequest()** function. It represents a request to a remote node to remove the back-up of an entry from the local primary binding table cache.

The `ZPS_tsAplZdpRemoveBkupBindEntryReq` structure is detailed below.

```
typedef struct {
   uint64 u64SrcAddress;
   uint8  u8SrcEndPoint;
   uint16 u16ClusterId;
   uint8  u8DstAddrMode;
   union {
      struct {
         uint16 u16DstAddress;
      } sShort;
      struct {
         uint64 u64DstAddress;
         uint8  u8DstEndPoint;
      } sExtended;
   };
} ZPS_tsAplZdpRemoveBkupBindEntryReq;
```

where:

- `u64SrcAddress` is the IEEE address of the source node for the binding entry
- `u8SrcEndpoint` is the number of the source endpoint for the binding (1-240)
- `u16ClusterId` is the ID of the cluster (on the local endpoint) for the binding
- `u8DstAddrMode` is the destination addressing mode for remaining elements (see Table 16 below)
- `u16DstAddress` is the address the destination node for the binding (address type according to setting of `u8DstAddrMode`)
- `u8DstEndPoint` is the number of the destination endpoint for the binding (1-240)

u8DstAddrMode	Code	Description
0x01	ZPS_E_ADDR_MODE_GROUP	16-bit Group address
0x03	ZPS_E_ADDR_MODE_IEEE	64-bit IEEE/MAC address

Table 16: Addressing Modes

8.2.2.29 ZPS_tsAplZdpBackupBindTableReq

This structure is used by the function **ZPS_eAplZdpBackupBindTableRequest()**. It represents a request to a remote node to save a back-up of the local primary binding table cache (whole or in part).

The `ZPS_tsAplZdpBackupBindTableReq` structure is detailed below.

```
typedef struct {
   uint16 u16BindingTableEntries;
   uint16 u16StartIndex;
   uint16 u16BindingTableListCount;
   /* Rest of message is variable length */
   ZPS_tsAplZdpBindingTable sBindingTable;
} ZPS_tsAplZdpBackupBindTableReq;
```

where:

- `u16BindingTableEntries` is the total number of entries in the primary binding table cache
- `u16StartIndex` is the binding table index of the first entry to be backed up
- `u16BindingTableListCount` is the number of binding table entries in the list to be backed up (`sBindingTable`)
- `sBindingTable` is a pointer to the list of binding table entries to be backed up. Each list item is of the type `ZPS_tsAplZdpBindingTable` detailed below

ZPS_tsAplZdpBindingTable

```
typedef struct
{
   uint64  u64SourceAddress;
   ZPS_tsAplZdpBindingTableEntry* psBindingTableEntryForSpSrcAddr;
}ZPS_tsAplZdpBindingTable;
```

where:

- `u64SourceAddress` is the IEEE source address for the binding table entry
- `psBindingTableEntryForSpSrcAddr` is the binding table entry. This is of the type `ZPS_tsAplZdpBindingTableEntry` detailed below

ZPS_tsAplZdpBindingTableEntry

```
typedef struct
{
   uint16  u16ClusterId;
   uint8   u8SourceEndpoint;
   uint8   u8DstAddrMode;
   union {
      struct {
         uint16 u16DstAddress;
      } sShort;
      struct {
         uint64 u64DstAddress;
         uint8  u8DstEndPoint;
      } sExtended;
   };
}ZPS_tsAplZdpBindingTableEntry;
```

where:

- `u16ClusterId` is the ID of the cluster (on the local endpoint) for the binding
- `u8SrcEndpoint` is the number of the source endpoint for the binding (1-240)
- `u8DstAddrMode` is the destination addressing mode for remaining elements (see Table 17 below)
- `u16DstAddress` is the address the destination node for the binding (address type according to setting of `u8DstAddrMode`)
- `u8DstEndPoint` is the number of the destination endpoint for the binding (1-240)

u8DstAddrMode	Code	Description
0x01	ZPS_E_ADDR_MODE_GROUP	16-bit Group address
0x03	ZPS_E_ADDR_MODE_IEEE	64-bit IEEE/MAC address

Table 17: Addressing Modes

8.2.2.30 ZPS_tsAplZdpRecoverBindTableReq

This structure is used by the function **ZPS_eAplZdpRecoverBindTableRequest()**. It represents a request to a remote node to recover a back-up of the local primary binding table cache.

The `ZPS_tsAplZdpRecoverBindTableReq` structure is detailed below.

```
typedef struct {
   uint16 u16StartIndex;
} ZPS_tsAplZdpRecoverBindTableReq;
```

where `u16StartIndex` is the binding table index of the first entry to be recovered.

8.2.2.31 ZPS_tsAplZdpBackupSourceBindReq

This structure is used by the function **ZPS_eAplZdpBackupSourceBindRequest()**. It represents a request to a remote node to save a back-up of the local node's source binding table (whole or in part).

The `ZPS_tsAplZdpBackupSourceBindReq` structure is detailed below.

```
typedef struct {
   uint16 u16SourceTableEntries;
   uint16 u16StartIndex;
   uint16 u16SourceTableListCount;
   /* Rest of message is variable length */
   uint64* pu64SourceAddress;
} ZPS_tsAplZdpBackupSourceBindReq;
```

where:

- `u16SourceTableEntries` is the total number of entries in the source binding table
- `u16StartIndex` is the binding table index of the first entry to be backed up
- `u16SourceTableListCount` is the number of binding table entries in the list to be backed up (`pu64SourceAddress`)
- `pu64SourceAddress` is a pointer to the list of IEEE source addresses corresponding to the binding table entries to be backed up

8.2.2.32 ZPS_tsAplZdpRecoverSourceBindReq

This structure is used by the function **ZPS_eAplZdpRecoverSourceBindRequest()**. It represents a request to a remote node to recover the back-up of the local node's source binding table (whole or in part).

The `ZPS_tsAplZdpRecoverSourceBindReq` structure is detailed below.

```
typedef struct {
   uint16 u16StartIndex;
} ZPS_tsAplZdpRecoverSourceBindReq;
```

where `u16StartIndex` is the binding table index of the first entry to be recovered.

8.2.2.33 ZPS_tsAplZdpMgmtNwkDiscReq

This structure is used by the function **ZPS_eAplZdpMgmtNwkDiscRequest()**. It represents a request to a remote node to discover any other wireless networks that are operating in the neighbourhood.

The `ZPS_tsAplZdpMgmtNwkDiscReq` structure is detailed below.

```
typedef struct {
    uint32 u32ScanChannels;
    uint8  u8ScanDuration;
    uint8  u8StartIndex;
} ZPS_tsAplZdpMgmtNwkDiscReq;
```

where:

- `u32ScanChannels` is a bitmask of the radio channels to scan ('1' means scan, '0' means do not scan):
  - Bits 0 to 26 respectively represent channels 0 to 26 (only bits 11 to 26 are relevant to the 2400-MHz band)
  - Bits 27 to 31 are reserved
- `u8ScanDuration` is a value in the range 0x00 to 0x0E that determines the time spent scanning each channel - this time is proportional to $2^{u8ScanDuration}+1$
- `u8StartIndex` is the index of the first result from the results list to include in the response to this request

8.2.2.34 ZPS_tsAplZdpMgmtLqiReq

This structure is used by the function **ZPS_eAplZdpMgmtLqiRequest()**. It represents a request to a remote node to provide a list of neighbouring nodes, from its Neighbour table, including a radio signal strength (LQI) value for each of these nodes.

The `ZPS_tsAplZdpMgmtLqiReq` structure is detailed below.

```
typedef struct {
    uint8  u8StartIndex;
} ZPS_tsAplZdpMgmtLqiReq;
```

where `u8StartIndex` is the Neighbour table index of the first entry to be included in the response to this request.

8.2.2.35 ZPS_tsAplZdpMgmtRtgReq

This structure is used by the function **ZPS_eAplZdpMgmtRtgRequest()**. It represents a request to a remote node to provide the contents of its Routing table.

The `ZPS_tsAplZdpMgmtRtgReq` structure is detailed below.

```
typedef struct {
   uint8  u8StartIndex;
} ZPS_tsAplZdpMgmtRtgReq;
```

where `u8StartIndex` is the Routing table index of the first entry to be included in the response to this request.

8.2.2.36 ZPS_tsAplZdpMgmtBindReq

This structure is used by the function **ZPS_eAplZdpMgmtBindRequest()**. It represents a request to a remote node to provide the contents of its Binding table.

The `ZPS_tsAplZdpMgmtBindReq` structure is detailed below.

```
typedef struct {
   uint8  u8StartIndex;
} ZPS_tsAplZdpMgmtBindReq;
```

where `u8StartIndex` is the Binding table index of the first entry to be included in the response to this request.

8.2.2.37 ZPS_tsAplZdpMgmtLeaveReq

This structure is used by the function **ZPS_eAplZdpMgmtLeaveRequest()**. It requests a remote node to leave the network.

The `ZPS_tsAplZdpMgmtLeaveReq` structure is detailed below.

```
typedef struct {
   uint64 u64DeviceAddress;
   uint8  u8Flags;
} ZPS_tsAplZdpMgmtLeaveReq;
```

where:

- `u64DeviceAddress` is the IEEE address of the device being asked to leave the network
- `u8Flags` is an 8-bit bitmap containing the following flags:
    - Rejoin flag (bit 0): Set to 1 if the node requested to leave the network should immediately try to rejoin the network, otherwise set to 0.
    - Remove Children flag (bit 1): Set to 1 if the node requested to leave the network should also request its own children (if any) to leave the network, otherwise set to 0.
    - Reserved (bits 7-2)

8.2.2.38 ZPS_tsAplZdpMgmtDirectJoinReq

This structure is used by the function **ZPS_eAplZdpMgmtDirectJoinRequest()**. It requests a remote node to allow a particular device to join it (and therefore the network).

The `ZPS_tsAplZdpMgmtDirectJoinReq` structure is detailed below.

```
typedef struct {
    uint64 u64DeviceAddress;
    uint8  u8Capability;
} ZPS_tsAplZdpMgmtDirectJoinReq;
```

where:

- `u64DeviceAddress` is the IEEE address of the device to be allowed to join
- `u8Capability` is a bitmask of the operating capabilities of the device to be allowed to join. This bitmask is detailed in Table 8 on page 199

8.2.2.39 ZPS_tsAplZdpMgmtPermitJoiningReq

This structure is used by the function **ZPS_eAplZdpMgmtPermitJoiningRequest()**. It requests a remote node (Router or Co-ordinator) to enable or disable joining for a specified amount of time.

The `ZPS_tsAplZdpMgmtPermitJoiningReq` structure is detailed below.

```
typedef struct {
    uint8  u8PermitDuration;
    bool_t bTcSignificance;
} ZPS_tsAplZdpMgmtPermitJoiningReq;
```

where:

- `u8PermitDuration` is the time period, in seconds, during which joining will be allowed (0x00 means that joining is enabled or disabled with no time limit)
- `bTcSignificance` determines whether the remote device is a ‘Trust Centre’:
    - TRUE: A Trust Centre
    - FALSE: Not a Trust Centre

8.2.2.40 ZPS_tsAplZdpMgmtCacheReq

This structure is used by the function **ZPS_eAplZdpMgmtCacheRequest()**. It requests a remote node to provide a list of the End Devices registered in its primary discovery cache.

The `ZPS_tsAplZdpMgmtCacheReq` structure is detailed below.

```
typedef struct {
    uint8 u8StartIndex;
} ZPS_tsAplZdpMgmtCacheReq;
```

where `u8StartIndex` is the discovery cache index of the first entry to be included in the response to this request.

8.2.2.41 ZPS_tsAplZdpMgmtNwkUpdateReq

This structure is used by the function **ZPS_eAplZdpMgmtNwkUpdateRequest()**. It requests an update of network parameters related to radio communication and may optionally initiate an energy scan in the 2400-MHz band.

The `ZPS_tsAplZdpMgmtNwkUpdateReq` structure is detailed below.

```
typedef struct {
   uint32 u32ScanChannels;
   uint8  u8ScanDuration;
   uint8  u8ScanCount;
   uint8  u8NwkUpdateId;
   uint16 u16NwkManagerAddr;
} ZPS_tsAplZdpMgmtNwkUpdateReq;
```

where:

- `u32ScanChannels` is a bitmask of the radio channels to be scanned ('1' means scan, '0' means do not scan):
  - Bits 0 to 26 respectively represent channels 0 to 26 (only bits 11 to 26 are relevant to the 2400-MHz band)
  - Bits 27 to 31 are reserved
- `u8ScanDuration` is a key value used to determine the action to be taken, as follows:
  - 0x00-0x05: Indicates that an energy scan is required and determines the time to be spent scanning each channel - this time is proportional to $2^{\texttt{u8ScanDuration}}+1$. The set of channels to scan is specified through `u32ScanChannels` and the maximum number of scans is equal to the value of `u8ScanCount`. Valid for unicasts only
  - 0x06-0xFD: Reserved
  - 0xFE: Indicates that radio channel is to be changed to single channel specified through `u32ScanChannels` and that network manager address to be set to that specified through `u16NwkManagerAddr`. Valid for broadcasts only
  - 0xFF: Indicates that stored radio channel mask to be updated with that specified through `u32ScanChannels` (but scan not required). Valid for broadcasts only.
- `u8ScanCount` is the number of energy scans to be conducted and reported. Valid only if a scan has been enabled through `u8ScanDuration` (0x00-0x05)
- `u8NwkUpdateId` is a value set by the Network Channel Manager before the request is sent. Valid only if `u8ScanDuration` set to 0xFE or 0xFF
- `u16NwkManagerAddr` is the 16-bit network address of the Network Manager (node nominated to manage radio-band operation of network). Valid only if `u8ScanDuration` set to 0xFF

8.2.2.42 ZPS_tsAplZdpParentAnnceReq

This structure is used by the function **ZPS_eAplZdpParentAnnceReq()**, which sends out a Parent Announcement message. The structure specifies the nodes that are the children of the local node which called the function.

The `ZPS_tsAplZdpParentAnnceReq` structure is detailed below.

```
typedef struct {
    uint8    u8NumberOfChildren;
    uint64*  pu64ChildList;
} ZPS_tsAplZdpParentAnnceReq;
```

where:

- `u8NumberOfChildren` is the number of child nodes
- `pu64ChildList` is a pointer to a list of the 64-bit IEEE/MAC addresses of the child nodes

8.2.3 ZDP Response Structures

This section details the structures that are used to store ZDP responses, resulting from requests sent using the ZDP functions. A received response is collected using the function **ZQ_bZQueueReceive()**. As part of this function call, you must provide a pointer to a structure to store the message data. This structure must be of the appropriate type for the response, from those described in this section.

The ZDP response structures are listed below, along with their page references.

8.2.3.1 ZPS_tsAplZdpNwkAddrRsp

This structure is used to store NWK_addr_rsp message data - a response to a call to the function **ZPS_eAplZdpNwkAddrRequest()**. This response contains the network address of the node with a given IEEE address.

The `ZPS_tsAplZdpNwkAddrRsp` structure is detailed below.

```
typedef struct {
    uint8  u8Status;
    uint64 u64IeeeAddrRemoteDev;
    uint16 u16NwkAddrRemoteDev;
    uint8  u8NumAssocDev;
    uint8  u8StartIndex;
    /* Rest of the message is variable Length */
    uint16* pNwkAddrAssocDevList;
} ZPS_tsAplZdpNwkAddrRsp;
```

where:

- `u8Status` is the return status for **ZPS_eAplZdpNwkAddrRequest()**
- `u64IeeeAddrRemoteDev` is the IEEE address of the remote node that sent the response (this is the IEEE address specified in the original request)
- `u16NwkAddrRemoteDev` is the network address of the remote node that sent the response (this is the network address that was requested)
- `u8NumAssocDev` is the number of neighbouring nodes for which network addresses are also being reported (in the remainder of the structure)
- `u8StartIndex` is the index in the remote node's Neighbour table of the first entry to be included in this report. This element should be ignored if the element `u8NumAssocDev` is 0.
- `pNwkAddrAssocDevList` is a pointer to a list of 16-bit network addresses of the remote node's neighbours (this is a variable-length list with four bytes per node). This element should be ignored if the element `u8NumAssocDev` is 0.

8.2.3.2 ZPS_tsAplZdpIeeeAddrRsp

This structure is used to store IEEE_addr_rsp message data - a response to a call to the function **ZPS_eAplZdpIeeeAddrRequest()**. This response contains the IEEE address of the node with a given network address.

The ZPS_tsAplZdpIeeeAddrRsp structure is detailed below.

```
typedef struct {
    uint8  u8Status;
    uint64 u64IeeeAddrRemoteDev;
    uint16 u16NwkAddrRemoteDev;
    uint8  u8NumAssocDev;
    uint8  u8StartIndex;
    /* Rest of the message is variable Length */
    uint16* pNwkAddrAssocDevList;
} ZPS_tsAplZdpIeeeAddrRsp;
```

where:

- u8Status is the return status for **ZPS_eAplZdpIeeeAddrRequest()**
- u64IeeeAddrRemoteDev is the IEEE address of the remote node that sent the response (this is the IEEE address that was requested)
- u16NwkAddrRemoteDev is the network address of the remote node that sent the response (this is the network address specified in the original request)
- u8NumAssocDev is the number of neighbouring nodes for which network addresses are also being reported (in the remainder of the structure)
- u8StartIndex is the index in the remote node's Neighbour table of the first entry to be included in this report. This element should be ignored if the element u8NumAssocDev is 0.
- pNwkAddrAssocDevList is a pointer to a list of 16-bit network addresses of the remote node's neighbours (this is a variable-length list with four bytes per node). This element should be ignored if the element u8NumAssocDev is 0.

8.2.3.3 ZPS_tsAplZdpNodeDescRsp

This structure is used to store Node_Desc_rsp message data - a response to a call to the function **ZPS_eAplZdpNodeDescRequest()**. This response contains the Node descriptor of the node with a given network address.

The `ZPS_tsAplZdpNodeDescRsp` structure is detailed below.

```
typedef struct {
   uint8  u8Status;
   uint16 u16NwkAddrOfInterest;
   /* Rest of the message is variable length */
   ZPS_tsAplZdpNodeDescriptor tsNodeDescriptor;
} ZPS_tsAplZdpNodeDescRsp;
```

where:

- `u8Status` is the return status for **ZPS_eAplZdpNodeDescRequest()**
- `u16NwkAddrOfInterest` is the network address of the remote node that sent the response (this is the network address that was specified in the request)
- `tsNodeDescriptor` is the returned Node descriptor, a structure of type `ZPS_tsAplZdpNodeDescriptor` (detailed in Section 8.2.1.1). This is only included if `u8Status` reports success

8.2.3.4 ZPS_tsAplZdpPowerDescRsp

This structure is used to store Power_Desc_rsp message data - a response to a call to the function **ZPS_eAplZdpPowerDescRequest()**. This response contains the Power descriptor of the node with a given network address.

The `ZPS_tsAplZdpPowerDescRsp` structure is detailed below.

```
typedef struct {
   uint8  u8Status;
   uint16 u16NwkAddrOfInterest;
   /* Rest of the message is variable length */
   ZPS_tsAplZdpNodePowerDescriptor sPowerDescriptor;
} ZPS_tsAplZdpPowerDescRsp;
```

where:

- `u8Status` is the return status for **ZPS_eAplZdpPowerDescRequest()**
- `u16NwkAddrOfInterest` is the network address of the remote node that sent the response (this is the network address that was specified in the request)
- `sPowerDescriptor` is the returned Power descriptor, a structure of type `ZPS_tsAplZdpNodePowerDescriptor` (detailed in Section 8.2.1.2). This is only included if `u8Status` reports success

8.2.3.5 ZPS_tsAplZdpSimpleDescRsp

This structure is used to store Simple_Desc_rsp message data - a response to a call to the function **ZPS_eAplZdpSimpleDescRequest()**. This response contains the Simple descriptor of a given endpoint on the node with a given network address.

The `ZPS_tsAplZdpSimpleDescRsp` structure is detailed below.

```
typedef struct {
   uint8  u8Status;
   uint16 u16NwkAddrOfInterest;
   uint8  u8Length;
   /* Rest of the message is variable length */
   ZPS_tsAplZdpSimpleDescType sSimpleDescriptor;
} ZPS_tsAplZdpSimpleDescRsp;
```

where:

- `u8Status` is the return status for **ZPS_eAplZdpSimpleDescRequest()**
- `u16NwkAddrOfInterest` is the network address of the remote node that sent the response (this is the network address that was specified in the request)
- `u8Length` is the length of the returned Simple descriptor, in bytes (depends on the number of clusters supported by the endpoint)
- `sSimpleDescriptor` is the returned Simple descriptor, a structure of type `ZPS_tsAplZdpSimpleDescType` (detailed in Section 8.2.1.3). This is only included if `u8Status` reports success

8.2.3.6 ZPS_tsAplZdpExtendedSimpleDescRsp

This structure is used to store Extended_Simple_Desc_rsp message data - a response to a call to the function **ZPS_eAplZdpExtendedSimpleDescRequest()**. This response contains a cluster list (combined input and output) for a given endpoint on the node with a given network address.

The `ZPS_tsAplZdpExtendedSimpleDescRsp` structure is detailed below.

```
typedef struct {
    uint8  u8Status;
    uint16 u16NwkAddr;
    uint8  u8EndPoint;
    uint8  u8AppInputClusterCount;
    uint8  u8AppOutputClusterCount;
    uint8  u8StartIndex;
    /* Rest of the message is variable length */
    uint16* pAppClusterList;
} ZPS_tsAplZdpExtendedSimpleDescRsp;
```

where:

- `u8Status` is the return status for **ZPS_eAplZdpExtendedSimpleDescRequest()**
- `u16NwkAddr` is the network address of the remote node that sent the response (this is the network address that was specified in the request)
- `u8EndPoint` is the number of the endpoint for which the response was sent (this is the endpoint number that was specified in the request)
- `u8AppInputClusterCount` is the total number of input clusters in the endpoint's complete input cluster list
- `u8AppOutputClusterCount` is the total number of output clusters in the endpoint's complete output cluster list
- `u8StartIndex` is the index, in the endpoint's complete input or output cluster list, of the first cluster reported in this response
- `pAppClusterList` is a pointer to the reported cluster list, input clusters first then output clusters. This is only included if `u8Status` reports success

8.2.3.7 ZPS_tsAplZdpComplexDescRsp

This structure is used to store Complex_Desc_rsp message data - a response to a call to the function **ZPS_eAplZdpComplexDescRequest()**. This response contains the Complex descriptor of the node with a given network address.

The `ZPS_tsAplZdpComplexDescRsp` structure is detailed below.

```
typedef struct {
   uint8  u8Status;
   uint16 u16NwkAddrOfInterest;
   uint8  u8Length;
   /* Rest of the message is variable Length */
   ZPS_tsAplZdpComplexDescElement sComplexDescriptor;
} ZPS_tsAplZdpComplexDescRsp;
```

where:

- `u8Status` is the return status for **ZPS_eAplZdpComplexDescRequest()**
- `u16NwkAddrOfInterest` is the network address of the remote node that sent the response (this is the network address that was specified in the request)
- `u8Length` is the length of the returned Complex descriptor, in bytes
- `sComplexDescriptor` is the returned Complex descriptor, a structure of type `ZPS_tsAplZdpComplexDescRsp` (described below). This is only included if `u8Status` reports success

ZPS_tsAplZdpComplexDescElement

```
typedef struct {
    uint8 u8XMLTag;
    uint8 u8FieldCount;
    uint8 *pu8Data;
} ZPS_tsAplZdpComplexDescElement;
```

where:

- `u8XMLTag` is the XML tag for the current field
- `u8FieldCount` is the number of fields in the Complex descriptor
- `*pu8Data` is a pointer to the data of the current field

8.2.3.8 ZPS_tsAplZdpUserDescRsp

This structure is used to store User_Desc_rsp message data - a response to a call to the function **ZPS_eAplZdpUserDescRequest()**. This response contains the User descriptor of the node with a given network address.

The `ZPS_tsAplZdpUserDescRsp` structure is detailed below.

```
typedef struct {
    uint8  u8Status;
    uint16 u16NwkAddrOfInterest;
    uint8  u8Length;
    /* Rest of the message is variable Length */
    char szUserDescriptor[ZPS_ZDP_LENGTH_OF_USER_DESC];
} ZPS_tsAplZdpUserDescRsp;
```

where:

- `u8Status` is the return status for **ZPS_eAplZdpUserDescRequest()**
- `u16NwkAddrOfInterest` is the network address of the remote node that sent the response (this is the network address that was specified in the request)
- `u8Length` is the length of the returned User descriptor, in bytes (maximum: 16)
- `szUserDescriptor` is the returned User descriptor as a character array. This is only included if `u8Status` reports success

8.2.3.9 ZPS_tsAplZdpMatchDescRsp

This structure is used to store Match_Desc_rsp message data - a response to a call to the function **ZPS_eAplZdpMatchDescRequest()**. This response contains details of the endpoints on the remote node that matched the criteria specified in the original request.

The `ZPS_tsAplZdpMatchDescRsp` structure is detailed below.

```
typedef struct {
   uint8  u8Status;
   uint16 u16NwkAddrOfInterest;
   uint8  u8MatchLength;
   /* Rest of message is variable length */
   uint8* u8MatchList;
} ZPS_tsAplZdpMatchDescRsp;
```

where:

- `u8Status` is the return status for **ZPS_eAplZdpMatchDescRequest()**
- `u16NwkAddrOfInterest` is the network address of the remote node that sent the response (this is the network address that was specified in the request)
- `u8MatchLength` is the length of the list of matched endpoints, in bytes
- `u8MatchList` is a pointer to the list of matched endpoints, where each endpoint is represented by an 8-bit value (in the range 1-240)

8.2.3.10 ZPS_tsAplZdpActiveEpRsp

This structure is used to store Active_EP_rsp message data - a response to a call to the function **ZPS_eAplZdpActiveEpRequest()**. This response contains a list of the active endpoints on a given network node.

The `ZPS_tsAplZdpActiveEpRsp` structure is detailed below.

```
typedef struct {
   uint8  u8Status;
   uint16 u16NwkAddrOfInterest;
   uint8  u8ActiveEpCount;
   /* Rest of the message is variable */
   uint8* pActiveEpList;
} ZPS_tsAplZdpActiveEpRsp;
```

where:

- `u8Status` is the return status for **ZPS_eAplZdpActiveEpRequest()**
- `u16NwkAddrOfInterest` is the network address of the remote node that sent the response (this is the network address that was specified in the request)
- `u8ActiveEpCount` is the number of active endpoints on the node
- `pActiveEpList` is a pointer to the list of active endpoints, where each endpoint is represented by an 8-bit value (in the range 1-240).

8.2.3.11 ZPS_tsAplZdpExtendedActiveEpRsp

This structure is used to store Extended_Active_EP_rsp message data - a response to a call to the function **ZPS_eAplZdpExtendedActiveEpRequest()**. This response contains a list of the active endpoints on the node with a given network address.

The `ZPS_tsAplZdpExtendedActiveEpRsp` structure is detailed below.

```
typedef struct {
   uint8  u8Status;
   uint16 u16NwkAddr;
   uint8  u8ActiveEpCount;
   uint8  u8StartIndex;
   /* Rest of the message is variable Length */
   uint8* pActiveEpList;
} ZPS_tsAplZdpExtendedActiveEpRsp;
```

where:

- `u8Status` is the return status for **ZPS_eAplZdpExtendedActiveEpRequest()**
- `16NwkAddr` is the network address of the remote node that sent the response (this is the network address that was specified in the request)
- `u8ActiveEpCount` is the total number of active endpoints on the node
- `u8StartIndex` is the index, in the node's list of active endpoints, of the first endpoint reported in this response
- `pActiveEpList` is a pointer to the reported list of active endpoints (starting with the endpoint with index `u8StartIndex`).

8.2.3.12 ZPS_tsAplZdpUserDescConf

This structure is used to store User_Desc_conf message data - a response to a call to the function **ZPS_eAplZdpUserDescSetRequest()**. This response contains a confirmation of the requested configuration of the User descriptor on a given network node.

The `ZPS_tsAplZdpUserDescConf` structure is detailed below.

```
typedef struct {
uint8  u8Status;
uint16 u16NwkAddrOfInterest;
} ZPS_tsAplZdpUserDescConf;
```

where:

- `u8Status` is the return status for **ZPS_eAplZdpUserDescSetRequest()**
- `u16NwkAddrOfInterest` is the network address of the remote node that sent the response (this is the network address that was specified in the request)

8.2.3.13 ZPS_tsAplZdpSystemServerDiscoveryRsp

This structure is used to store System_Server_Discovery_rsp message data - a response to a call to the function **ZPS_eAplZdpSystemServerDiscoveryRequest()**. This response indicates which of the requested services are supported by a given network node.

The `ZPS_tsAplZdpSystemServerDiscoveryRsp` structure is detailed below.

```
typedef struct {
    uint8  u8Status;
    uint16 u16ServerMask;
} ZPS_tsAplZdpSystemServerDiscoveryRsp;
```

where:

- `u8Status` is the return status for the function **ZPS_eAplZdpSystemServerDiscoveryRequest()**
- `u16ServerMask` is the returned bitmask that summarises the requested services supported by the node (1 for 'supported', 0 for 'not supported' or 'not requested'). This bitmask is detailed in the table below.

Bit	Service
0	Primary Trust Centre
1	Backup Trust Centre
2	Primary Binding Table Cache
3	Backup Binding Table Cache
4	Primary Discovery Cache
5	Back-up Discovery Cache
6	Network Manager
7-15	Reserved

Table 18: Services Bitmask

8.2.3.14 ZPS_tsAplZdpDiscoveryCacheRsp

This structure is used to store Discovery_Cache_rsp message data - a response to a call to the function **ZPS_eAplZdpDiscoveryCacheRequest()**. This response indicates that the sending node has a primary discovery cache.

The `ZPS_tsAplZdpDiscoveryCacheRsp` structure is detailed below.

```
typedef struct {
    uint8  u8Status;
} ZPS_tsAplZdpDiscoveryCacheRsp;
```

where `u8Status` is the return status for **ZPS_eAplZdpDiscoveryCacheRequest()**.

8.2.3.15 ZPS_tsAplZdpDiscoveryStoreRsp

This structure is used to store Discovery_Store_rsp message data - a response to a call to the function **ZPS_eAplZdpDiscoveryStoreRequest()**. This response indicates whether the sending node has successfully reserved space in its primary discovery cache.

The `ZPS_tsAplZdpDiscoveryStoreRsp` structure is detailed below.

```
typedef struct {
    uint8   u8Status;
} ZPS_tsAplZdpDiscoveryStoreRsp;
```

where `u8Status` is the return status for **ZPS_eAplZdpDiscoveryStoreRequest()**.

8.2.3.16 ZPS_tsAplZdpNodeDescStoreRsp

This structure is used to store Node_Desc_store_rsp message data - a response to a call to the function **ZPS_eAplZdpNodeDescStoreRequest()**. This response indicates whether the sending node has successfully stored the received Node descriptor in its primary discovery cache.

The `ZPS_tsAplZdpNodeDescStoreRsp` structure is detailed below.

```
typedef struct {
    uint8  u8Status;
} ZPS_tsAplZdpNodeDescStoreRsp;
```

where `u8Status` is the return status for **ZPS_eAplZdpNodeDescStoreRequest()**.

8.2.3.17 ZPS_tsAplZdpPowerDescStoreRsp

This structure is used to store Power_Desc_store_rsp message data - a response to a call to the function **ZPS_eAplZdpPowerDescStoreRequest()**. This response indicates whether the sending node has successfully stored the received Power descriptor in its primary discovery cache.

The `ZPS_tsAplZdpPowerDescStoreRsp` structure is detailed below.

```
typedef struct {
     uint8  u8Status;
     uint64 u64IeeeAddr;
     /* Rest of message is variable length */
     ZPS_tsAplZdpNodePowerDescriptor sPowerDescriptor;
} ZPS_tsAplZdpPowerDescStoreRsp;
```

where:

- `u8Status` is the return status for **ZPS_eAplZdpPowerDescStoreRequest()**.
- `u64IeeeAddr` is the IEEE/MAC address of the device whose Power descriptor has been stored in the primary discovery cache.
- `sPowerDescriptor` is the Power descriptor stored (see Section 8.2.1.1).

8.2.3.18 ZPS_tsAplZdpSimpleDescStoreRsp

This structure is used to store Power_Desc_store_rsp message data - a response to a call to the function **ZPS_eAplZdpSimpleDescStoreRequest()**. This response indicates whether the sending node has successfully stored the received Simple descriptor in its primary discovery cache.

The `ZPS_tsAplZdpSimpleDescStoreRsp` structure is detailed below.

```
typedef struct {
    uint8  u8Status;
} ZPS_tsAplZdpSimpleDescStoreRsp;
```

where `u8Status` is the return status for **ZPS_eAplZdpSimpleDescStoreRequest()**.

8.2.3.19 ZPS_tsAplZdpActiveEpStoreRsp

This structure is used to store Active_EP_store_rsp message data - a response to a call to the function **ZPS_eAplZdpActiveEpStoreRequest()**. This response indicates whether the sending node has successfully stored the received list of active endpoints in its primary discovery cache.

The `ZPS_tsAplZdpActiveEpStoreRsp` structure is detailed below.

```
typedef struct {
    uint8   u8Status;
} ZPS_tsAplZdpActiveEpStoreRsp;
```

where `u8Status` is the return status for **ZPS_eAplZdpActiveEpStoreRequest()**.

8.2.3.20 ZPS_tsAplZdpFindNodeCacheRsp

This structure is used to store Find_node_cache_rsp message data - a response to a call to the function **ZPS_eAplZdpFindNodeCacheRequest()**. This response indicates that the sending node holds ‘discovery information’ about a given network node in its primary discovery cache.

The `ZPS_tsAplZdpFindNodeCacheRsp` structure is detailed below.

```
typedef struct {
    uint16 u16CacheNwkAddr;
    uint16 u16NwkAddr;
    uint64 u64IeeeAddr;
} ZPS_tsAplZdpFindNodeCacheRsp;
```

where:

- `u16CacheNwkAddr` is the network address of the remote node that sent the response
- `u16NwkAddr` is the network address of the node of interest (this is the network address that was specified in the request)
- `u64IeeeAddr` is the IEEE address of the node of interest (this is the IEEE address that was specified in the request)

8.2.3.21 ZPS_tsAplZdpRemoveNodeCacheRsp

This structure is used to store Remove_node_cache_rsp message data - a response to a call to the function **ZPS_eAplZdpRemoveNodeCacheRequest()**. This response indicates whether the sending node has successfully removed from its primary discovery cache all 'discovery information' relating to a given End Device node.

The `ZPS_tsAplZdpRemoveNodeCacheRsp` structure is detailed below.

```
typedef struct {
    uint8  u8Status;
} ZPS_tsAplZdpRemoveNodeCacheRsp;
```

where `u8Status` is the return status for the function **ZPS_eAplZdpRemoveNodeCacheRequest()**.

8.2.3.22 ZPS_tsAplZdpEndDeviceBindRsp

This structure is used to store End_Device_Bind_rsp message data - a response to a call to the function **ZPS_eAplZdpEndDeviceBindRequest()**. This response is issued by the Co-ordinator to indicate the status of an End Device binding request.

The `ZPS_tsAplZdpEndDeviceBindRsp` structure is detailed below.

```
typedef struct {
    uint8 u8Status;
} ZPS_tsAplZdpEndDeviceBindRsp;
```

where `u8Status` is the return status for **ZPS_eAplZdpEndDeviceBindRequest()**.

8.2.3.23 ZPS_tsAplZdpBindRsp

This structure is used to store Bind_rsp message data - a response to a call to the function **ZPS_eAplZdpBindUnbindRequest()**. This response indicates the status of a binding request (a request to modify of a binding table).

The `ZPS_tsAplZdpBindRsp` structure is detailed below.

```
typedef struct {
    uint8 u8Status;
} ZPS_tsAplZdpBindRsp;
```

where `u8Status` is the return status for **ZPS_eAplZdpBindUnbindRequest()**.

8.2.3.24 ZPS_tsAplZdpUnbindRsp

This structure is used to store Unbind_rsp message data - a response to a call to the function **ZPS_eAplZdpBindUnbindRequest()**. This response indicates the status of an unbinding request (a request to modify of a binding table).

The `ZPS_tsAplZdpUnbindRsp` structure is detailed below.

```
typedef struct {
   uint8 u8Status;
} ZPS_tsAplZdpUnbindRsp;
```

where `u8Status` is the return status for **ZPS_eAplZdpBindUnbindRequest()**.

8.2.3.25 ZPS_tsAplZdpBindRegisterRsp

This structure is used to store Bind_Register_rsp message data - a response to a call to the function **ZPS_eAplZdpBindRegisterRequest()**. This response contains binding information held on the responding node concerning the requesting node.

The `ZPS_tsAplZdpBindRegisterRsp` structure is detailed below.

```
typedef struct {
   uint8  u8Status;
   uint16 u16BindingTableEntries;
   uint16 u16BindingTableListCount;
   /* Rest of the message is variable Length */
   ZPS_tsAplZdpBindingTable sBindingTableList;
} ZPS_tsAplZdpBindRegisterRsp;
```

where:

- `u8Status` is the return status for **ZPS_eAplZdpBindRegisterRequest()**
- `u16BindingTableEntries` is the total number of binding table entries concerning the requesting node held on the responding node
- `u16BindingTableListCount` is the number of binding table entries concerning the requesting node contained in this response
- `sBindingTableList` is a pointer to the first item in the list of reported binding table entries. A list item is of type `ZPS_tsAplZdpBindingTable` detailed below

ZPS_tsAplZdpBindingTable

```
typedef struct
{
   uint64  u64SourceAddress;
   ZPS_tsAplZdpBindingTableEntry* psBindingTableEntryForSpSrcAddr;
}ZPS_tsAplZdpBindingTable;
```

where:

- `u64SourceAddress` is the IEEE address of the node to which the binding table entry relates
- `psBindingTableEntryForSpSrcAddr` is a pointer to the relevant binding table information. This information is contained in a structure of type `ZPS_tsAplZdpBindingTableEntry` detailed below

ZPS_tsAplZdpBindingTableEntry

```
typedef struct
{
    uint8   u8SourceEndpoint;
    uint16  u16ClusterId;
    uint8   u8DstAddrMode;
    union {
        struct {
            uint16 u16DstAddress;
        } sShort;
        struct {
            uint64 u64DstAddress;
            uint8  u8DstEndPoint;
        } sExtended;
    };
}ZPS_tsAplZdpBindingTableEntry;
```

where:

- `u8SourceEndpoint` is the number of the bound endpoint (1-240) on the source node of the binding
- `u16ClusterId` is the ID of the cluster involved in the binding, on the source node of the binding
- `u8DstAddrMode` is the addressing mode used in the rest of the structure (see Table 19 below)
- `u16DstAddress` is the network address of the destination node of the binding (this is only application if `u8DstAddrMode` is set to 0x03)
- `u64DstAddress` is the IEEE address of the destination node of the binding (this is only application if `u8DstAddrMode` is set to 0x04)
- `u8DstEndPoint` is the number of the bound endpoint (1-240) on the destination node of the binding

u8DstAddrMode	Code	Description
0x00	ZPS_E_ADDR_MODE_BOUND	Bound endpoint
0x01	ZPS_E_ADDR_MODE_GROUP	16-bit Group address
0x02	ZPS_E_ADDR_MODE_SHORT	16-bit Network (Short) address
0x03	ZPS_E_ADDR_MODE_IEEE	64-bit IEEE/MAC address

Table 19: Addressing Modes

8.2.3.26 ZPS_tsAplZdpReplaceDeviceRsp

This structure is used to store Replace_Device_rsp message data - a response to a call to the function **ZPS_eAplZdpReplaceDeviceRequest()**. This response indicates the status of the replace request.

The `ZPS_tsAplZdpReplaceDeviceRsp` structure is detailed below.

```
typedef struct {
   uint8 u8Status;
} ZPS_tsAplZdpReplaceDeviceRsp;
```

where `u8Status` is the return status for **ZPS_eAplZdpReplaceDeviceRequest()**.

8.2.3.27 ZPS_tsAplZdpStoreBkupBindEntryRsp

This structure is used to store Store_Bkup_Bind_Entry_rsp message data - a response to a call to the function **ZPS_eAplZdpStoreBkupBindEntryRequest()**. This response indicates the status of the back-up request.

The `ZPS_tsAplZdpStoreBkupBindEntryRsp` structure is detailed below.

```
typedef struct {
    uint8   u8Status;
} ZPS_tsAplZdpStoreBkupBindEntryRsp;
```

where `u8Status` is the return status for the function **ZPS_eAplZdpStoreBkupBindEntryRequest()**.

8.2.3.28 ZPS_tsAplZdpRemoveBkupBindEntryRsp

This structure is used to store Remove_Bkup_Bind_Entry_rsp message data - a response to a call to the function **ZPS_eAplZdpRemoveBkupBindEntryRequest()**. This response indicates the status of the remove request.

The `ZPS_tsAplZdpRemoveBkupBindEntryRsp` structure is detailed below.

```
typedef struct {
   uint8   u8Status;
} ZPS_tsAplZdpRemoveBkupBindEntryRsp;
```

where `u8Status` is the return status for the function **ZPS_eAplZdpRemoveBkupBindEntryRequest()**.

8.2.3.29 ZPS_tsAplZdpBackupBindTableRsp

This structure is used to store Backup_Bind_Table_rsp message data - a response to a call to the function **ZPS_eAplZdpBackupBindTableRequest()**. This response indicates the status of the back-up request.

The `ZPS_tsAplZdpBackupBindTableRsp` structure is detailed below.

```
typedef struct {
    uint8   u8Status;
    uint16  u16EntryCount;
} ZPS_tsAplZdpBackupBindTableRsp;
```

where:

- `u8Status` is the return status for **ZPS_eAplZdpBackupBindTableRequest()**
- `u16EntryCount` is the number of binding table entries that have been backed up

8.2.3.30 ZPS_tsAplZdpRecoverBindTableRsp

This structure is used to store Recover_Bind_Table_rsp message data - a response to a call to the function **ZPS_eAplZdpRecoverBindTableRequest()**. This response indicates the status of the recover request and contains the recovered binding table entries.

The `ZPS_tsAplZdpRecoverBindTableRsp` structure is detailed below.

```
typedef struct {
   uint8   u8Status;
   uint16  u16StartIndex;
   uint16  u16BindingTableEntries;
   uint16  u16BindingTableListCount;
   /* Rest of the message is variable length */
   ZPS_tsAplZdpBindingTable sBindingTableList;
} ZPS_tsAplZdpRecoverBindTableRsp;
```

where:

- `u8Status` is the return status for **ZPS_eAplZdpRecoverBindTableRequest()**
- `u16StartIndex` is the binding table index of the first entry in the set of recovered binding table entries (`sBindingTableList`)
- `u16BindingTableEntries` is the total number of entries in the back-up binding table cache
- `u16BindingTableListCount` is the number of entries in the set of recovered binding table entries (`sBindingTableList`)
- `sBindingTableList` is a pointer to the first item in the list of recovered binding table entries. A list item is of type `ZPS_tsAplZdpBindingTable`, detailed in Section 8.2.3.25

8.2.3.31 ZPS_tsAplZdpBackupSourceBindRsp

This structure is used to store Backup_Source_Bind_rsp message data - a response to a call to the function **ZPS_eAplZdpBackupSourceBindRequest()**. This response indicates the status of the back-up request.

The `ZPS_tsAplZdpBackupSourceBindRsp` structure is detailed below.

```
typedef struct {
   uint8  u8Status;
} ZPS_tsAplZdpBackupSourceBindRsp;
```

where `u8Status` is the return status for the function **ZPS_eAplZdpBackupSourceBindRequest()**.

8.2.3.32 ZPS_tsAplZdpRecoverSourceBindRsp

This structure is used to store Recover_Source_Bind_rsp message data - a response to a call to the function **ZPS_eAplZdpRecoverSourceBindRequest()**. This response indicates the status of the recover request and contains the recovered binding table entries.

The `ZPS_tsAplZdpRecoverSourceBindRsp` structure is detailed below.

```
typedef struct {
   uint8  u8Status;
   uint16 u16StartIndex;
   uint16 u16SourceTableEntries;
   uint16 u16SourceTableListCount;
   /* Rest of the message is variable length */
   uint64* pu64SourceTableList;
} ZPS_tsAplZdpRecoverSourceBindRsp;
```

where:

- `u8Status` is the return status for the function **ZPS_eAplZdpRecoverSourceBindRequest()**
- `u16StartIndex` is the binding table index of the first entry in the set of recovered binding table entries (`pu64SourceTableList`)
- `u16SourceTableEntries` is the total number of source binding table entries in the back-up binding table cache
- `u16SourceTableListCount` is the number of entries in the set of recovered binding table entries (`pu64SourceTableList`)
- `pu64SourceTableList` is a pointer to the first item in the list of recovered binding table entries

8.2.3.33 ZPS_tsAplZdpMgmtNwkDiscRsp

This structure is used to store Mgmt_NWK_Disc_rsp message data - a response to a call to the function **ZPS_eAplZdpMgmtNwkDiscRequest()**. This response reports the networks discovered in a network discovery (all the networks or a subset).

The `ZPS_tsAplZdpMgmtNwkDiscRsp` structure is detailed below.

```
typedef struct {
   uint8 u8Status;
   uint8 u8NetworkCount;
   uint8 u8StartIndex;
   uint8 u8NetworkListCount;
   /* Rest of the message is variable length */
   ZPS_tsAplZdpNetworkDescr* psNetworkDescrList;
} ZPS_tsAplZdpMgmtNwkDiscRsp;
```

where:

- `u8Status` is the return status for **ZPS_eAplZdpMgmtNwkDiscRequest()**
- `u8NetworkCount` is the total number of networks discovered
- `u8StartIndex` is the index, in the complete list of discovered networks, of the first network reported in this response (through `psNetworkDescrList`)
- `u8NetworkListCount` is the number of discovered networks reported in this response (through `psNetworkDescrList`)
- `psNetworkDescrList` is a pointer to the first entry in a list of network descriptors for the discovered networks. Each entry is of the type `ZPS_tsAplZdpNetworkDescr` detailed below

ZPS_tsAplZdpNetworkDescr

```
typedef struct
{
    uint64 u64ExtPanId;
    uint8  u8LogicalChan;
    uint8  u8StackProfile;
    uint8  u8ZigBeeVersion;
    uint8  u8PermitJoining;
    uint8  u8RouterCapacity;
    uint8  u8EndDeviceCapacity;
} ZPS_tsAplZdpNetworkDescr;
```

where:

- `u64ExtPanId` is the 64-bit extended PAN ID of the discovered network
- `u8LogicalChan` is the radio channel in which the discovered network operates (value in range 0 to 26, but only channels 11 to 26 relevant to 2400-MHz band)
- `u8StackProfile` is the 4-bit identifier of the ZigBee stack profile used by the discovered network (0 - manufacturer-specific, 1 - ZigBee, 2 - ZigBee PRO, other values reserved) and is fixed at 2 for the NXP stack

- `u8ZigBeeVersion` is the 4-bit version of the ZigBee protocol used by the discovered network
- `u8PermitJoining` indicates whether the discovered network is currently allowing joinings - that is, at least one node (a Router or the Co-ordinator) of the network is allowing other nodes to join it:
  - 0x01: Joinings allowed
  - 0x00: Joinings not allowed
  - All other values reserved
- `u8RouterCapacity` indicates whether the device is capable of accepting join requests from Routers - set to TRUE if capable, FALSE otherwise
- `u8EndDeviceCapacity` indicates whether the device is capable of accepting join requests from End Devices - set to TRUE capable, FALSE otherwise

8.2.3.34 ZPS_tsAplZdpMgmtLqiRsp

This structure is used to store Mgmt_Lqi_rsp message data - a response to a call to the function **ZPS_eAplZdpMgmtLqiRequest()**. This response reports a list of neighbouring nodes along with their LQI (link quality) values.

The `ZPS_tsAplZdpMgmtLqiRsp` structure is detailed below.

```
typedef struct {
    uint8  u8Status;
    uint8  u8NeighborTableEntries;
    uint8  u8StartIndex;
    uint8  u8NeighborTableListCount;
    /* Rest of the message is variable length */
    ZPS_tsAplZdpNtListEntry* pNetworkTableList;
} ZPS_tsAplZdpMgmtLqiRsp;
```

where:

- `u8Status` is the return status for **ZPS_eAplZdpMgmtLqiRequest()**
- `u8NeighborTableEntries` is the total number of Neighbour table entries on the remote node
- `u8StartIndex` is the Neighbour table index of the first entry reported in this response (through `pNetworkTableList`)
- `u8NetworkListCount` is the number of Neighbour table entries reported in this response (through `pNetworkTableList`)
- `pNetworkTableList` is a pointer to the first entry in the list of reported Neighbour table entries. Each entry is of the type `ZPS_tsAplZdpNtListEntry` detailed below

ZPS_tsAplZdpNtListEntry

```
typedef struct
{
    uint64 u64ExtPanId;
    uint64 u64ExtendedAddress;
    uint16 u16NwkAddr;
    uint8  u8LinkQuality;
    uint8  u8Depth;
    /*
     * Bitfields are used for syntactic neatness and space saving.
     * May need to assess whether these are suitable for embedded
environment and may need to watch endianness on u8Assignment
     */
    union
    {
        struct
        {
            unsigned u1Reserved1:1;
            unsigned u2Relationship:3;
            unsigned u2RxOnWhenIdle:2;
            unsigned u2DeviceType:2;
            unsigned u6Reserved2:6;
            unsigned u2PermitJoining:2;
        } ;
        uint8 au8Field[2];
    } uAncAttrs;
} ZPS_tsAplZdpNtListEntry;
```

where:

- `u64ExtPanId` is the 64-bit extended PAN ID of the network
- `u64ExtendedAddress` is the IEEE address of the neighbouring node
- `u16NwkAddr` is the network address of the neighbouring node
- `u8LinkQuality` is the estimated LQI (link quality) value for radio transmissions from the neighbouring node
- `u8Depth` is the tree depth of the neighbouring node (where the Co-ordinator is at depth zero)
- `u1Reserved1:1` is a 1-bit reserved value and should be set zero.
- `u2Relationship:3` is a 3-bit value representing the neighbouring node's relationship to the local node:
  - 0: Neighbour is the parent
  - 1: Neighbour is a child
  - 2: Neighbour is a sibling (has same parent)
  - 3: None of the above

- 4: Neighbour is a former child
- `u2RxOnWhenIdle:2` is a 2-bit value indicating whether the neighbouring node's receiver is enable during idle periods:
  - 0: Receiver off when idle (sleeping device)
  - 1: Receiver on when idle (non-sleeping device)
  - 2: Unknown
- `u2DeviceType:2` is a 2-bit value representing the ZigBee device type of the neighbouring node:
  - 0: Co-ordinator
  - 1: Router
  - 2: End Device
  - 3: Unknown
- `u6Reserved2:6` is a 6-bit reserved value and should be set zero.
- `u2PermitJoining:2` is a 2-bit value indicating whether the neighbouring node is accepting joining requests:
  - 0: Not accepting join requests
  - 1: Accepting join requests
  - 2: Unknown
- `au8Field[2]` is the allocation of two bytes for the union.

8.2.3.35 ZPS_tsAplZdpMgmtRtgRsp

This structure is used to store Mgmt_Rtg_rsp message data - a response to a call to the function **ZPS_eAplZdpMgmtRtgRequest()**. This response reports the contents of the remote node's Routing table

The `ZPS_tsAplZdpMgmtRtgRsp` structure is detailed below.

```
typedef struct {
    uint8  u8Status;
    uint8  u8RoutingTableEntries;
    uint8  u8StartIndex;
    uint8  u8RoutingTableCount;
    /* Rest of the message is variable length */
    ZPS_tsAplZdpRtEntry* pRoutingTableList;
} ZPS_tsAplZdpMgmtRtgRsp;
```

where:

- `u8Status` is the return status for **ZPS_eAplZdpMgmtRtgRequest()**
- `u8RoutingTableEntries` is the total number of Routing table entries on the remote node
- `u8StartIndex` is the Routing table index of the first entry reported in this response (through `pRoutingTableList`)

- `u8RoutingTableCount` is the number of Routing table entries reported in this response (through `pRoutingTableList`)
- `pRoutingTableList` is a pointer to the first entry in the list of reported Routing table entries. Each entry is of the type `ZPS_tsAplZdpRtEntry` detailed below

ZPS_tsAplZdpRtEntry

```
typedef struct
{
   uint16 u16NwkDstAddr;      /**< Destination Network address */
   uint16 u16NwkNxtHopAddr;   /**< Next hop Network address */
   union
   {
      struct
      {
         unsigned u3Status:3;
         unsigned u1MemConst:1;
         unsigned u1ManyToOne:1;
         unsigned u1RouteRecordReqd:1;
         unsigned u1Reserved:2;
      } bfBitfields;
        uint8 u8Field;
    } uAncAttrs;
} ZPS_tsAplZdpRtEntry;
```

where:

- `u16NwkDstAddr` is the destination network address of the route
- `u16NwkNxtHopAddr` is the ‘next hop’ network address of the route
- `u3Status:3` is the 3-bit status for the route:
  - 000 = ACTIVE
  - 001 = DISCOVERY_UNDERWAY
  - 010 = DISCOVERY_FAILED
  - 011 = INACTIVE
  - 100 = VALIDATION_UNDERWAY
  - 101-111 = Reserved
- `u1MemConst:1` is a bit indicating whether the device is a memory-constrained concentrator
- `u1ManyToOne:1` is a bit indicating whether the destination node is a concentrator that issued a many-to-one request
- `u1RouteRecordReqd:1` is a bit indicating whether a route record command frame should be sent to the destination before the next data packet
- `u1Reserved:2` are reserved bits

- `u8Field` contains the full set of flags of the `bfBitfields` sub-structure, with `u3Status:3` occupying the most significant bits and `u1Reserved:2` occupying the least significant bits (for a big-endian device)

8.2.3.36 ZPS_tsAplZdpMgmtBindRsp

This structure is used to store Mgmt_Bind_rsp message data - a response to a call to the function **ZPS_eAplZdpMgmtBindRequest()**. This response reports the contents of the remote node's Binding table.

The `ZPS_tsAplZdpMgmtBindRsp` structure is detailed below.

```
typedef struct {
    uint8  u8Status;
    uint16 u16BindingTableEntries;
    uint16 u16StartIndex;
    uint16 u16BindingTableListCount;
    /* Rest of the message is variable length */
    ZPS_tsAplZdpBindingTable sBindingTableList;
} ZPS_tsAplZdpMgmtBindRsp;
```

where:

- `u8Status` is the return status for **ZPS_eAplZdpMgmtBindRequest()**
- `u16BindingTableEntries` is the total number of Binding table entries on the remote node
- `u8StartIndex` is the Binding table index of the first entry reported in this response (through `sBindingTableList`)
- `u16BindingTableListCount` is the number of Binding table entries reported in this response (through `sBindingTableList`)
- `sBindingTableList` is a pointer to the first entry in the list of reported Binding table entries. Each entry is of the type `ZPS_tsAplZdpBindingTable`, detailed in Section 8.2.2.29

8.2.3.37 ZPS_tsAplZdpMgmtLeaveRsp

This structure is used to store Mgmt_Leave_rsp message data - a response to a call to the function **ZPS_eAplZdpMgmtLeaveRequest()**. This response is issued by a remote node that has been requested to leave the network.

The `ZPS_tsAplZdpMgmtLeaveRsp` structure is detailed below.

```
typedef struct {
    uint8  u8Status;
} ZPS_tsAplZdpMgmtLeaveRsp;
```

where `u8Status` is the return status for **ZPS_eAplZdpMgmtLeaveRequest()**.

8.2.3.38 ZPS_tsAplZdpMgmtDirectJoinRsp

This structure is used to store Mgmt_Direct_Join_rsp message data - a response to a call to the function **ZPS_eAplZdpMgmtDirectJoinRequest()**. This response is issued by a remote node (Router or Co-ordinator) that has been requested to allow a particular device to join the network as a child of the node.

The `ZPS_tsAplZdpMgmtDirectJoinRsp` structure is detailed below.

```
typedef struct {
    uint8  u8Status;
} ZPS_tsAplZdpMgmtDirectJoinRsp;
```

where `u8Status` is the return status for **ZPS_eAplZdpMgmtDirectJoinRequest()**.

8.2.3.39 ZPS_tsAplZdpMgmtPermitJoiningRsp

This structure is used to store Mgmt_Permit_Joining_rsp message data - a response to a call to the function **ZPS_eAplZdpMgmtPermitJoiningRequest()**. This response is issued by a remote node (Router or Co-ordinator) that has been requested to enable or disable joining for a specified amount of time. The response is only sent if the original request was unicast (and not if it was broadcast).

The `ZPS_tsAplZdpMgmtPermitJoiningRsp` structure is detailed below.

```
typedef struct {
    uint8  u8Status;
} ZPS_tsAplZdpMgmtPermitJoiningRsp;
```

where `u8Status` is the return status for the function **ZPS_eAplZdpMgmtPermitJoiningRequest()**.

8.2.3.40 ZPS_tsAplZdpMgmtCacheRsp

This structure is used to store Mgmt_Cache_rsp message data - a response to a call to the function **ZPS_eAplZdpMgmtCacheRequest()**. This response reports a list of the End Devices registered in the node's primary discovery cache.

The `ZPS_tsAplZdpMgmtCacheRsp` structure is detailed below.

```
typedef struct {
   uint8 u8Status;
   uint8 u8DiscoveryCacheEntries;
   uint8 u8StartIndex;
   uint8 u8DiscoveryCacheListCount;
   /* Rest of the message is variable length */
   ZPS_tsAplDiscoveryCache* pDiscoveryCacheList;
} ZPS_tsAplZdpMgmtCacheRsp;
```

where:

- `u8Status` is the return status for **ZPS_eAplZdpMgmtCacheRequest()**
- `u8DiscoveryCacheEntries` is the total number of discovery cache entries on the remote node
- `u8StartIndex` is the discovery cache index of the first entry reported in this response (through `pDiscoveryCacheList`)
- `u8DiscoveryCacheListCount` is the number of discovery cache entries reported in this response (through `pDiscoveryCacheList`)
- `pRoutingTableList` is a pointer to the first entry in the list of reported discovery cache entries. Each entry is of the type `ZPS_tsAplDiscoveryCache` detailed below

ZPS_tsAplDiscoveryCache

```
typedef struct {
   uint64 u64ExtendedAddress;
   uint16 u16NwkAddress;
} ZPS_tsAplDiscoveryCache;
```

where:

- `u64ExtendedAddress` is the IEEE address of the End Device
- `u16NwkAddress` is the network address of the End Device

8.2.3.41 ZPS_tsAplZdpMgmtNwkUpdateNotify

This structure is used to store Mgmt_NWK_Update_notify message data - a notification which can be sent in response to a call to the function **ZPS_eAplZdpMgmtNwkUpdateRequest()**. This notification reports the results of an energy scan on the wireless network radio channels.

The `ZPS_tsAplZdpMgmtNwkUpdateNotify` structure is detailed below.

```
typedef struct {
   uint8  u8Status;
   uint32 u32ScannedChannels;
   uint16 u16TotalTransmissions;
   uint16 u16TransmissionFailures;
   uint8  u8ScannedChannelListCount;
   /* Rest of the message is variable Length */
   uint8*  u8EnergyValuesList;
} ZPS_tsAplZdpMgmtNwkUpdateNotify;
```

where:

- `u8Status` is the return status for **ZPS_eAplZdpMgmtNwkUpdateRequest()**
- `u32ScannedChannels` is a bitmask of the set of scanned radio channels ('1' means scanned, '0' means not scanned):
  - Bits 0 to 26 respectively represent channels 0 to 26 (only bits 11 to 26 are relevant to the 2400-MHz band)
  - Bits 27 to 31 are reserved
- `u16TotalTransmissions` is the total number of transmissions (from other networks) detected during the scan
- `u16TransmissionFailures` is the number of failed transmissions detected during the scan
- `u8ScannedChannelListCount` is the number of energy-level measurements (one per scanned channel) reported in this notification (through `u8EnergyValuesList`)
- `u8EnergyValuesList` is a pointer to the first in the set of reported energy-level measurements (the value 0xFF indicates there is too much interference on the channel)

8.2.3.42 ZPS_tsAplZdpParentAnnceRsp

This structure is used to store the data for a response to a Parent Announcement message that was sent using the function **ZPS_eAplZdpParentAnnceReq()**. This response reports any child nodes of the responding node that conflict with child nodes specified in the received Parent Announcement message.

The `ZPS_tsAplZdpParentAnnceRsp` structure is detailed below.

```
typedef struct {
    uint64* pu64ChildList;
    uint8   u8NumberOfChildren;
    uint8   u8Status;
} ZPS_tsAplZdpParentAnnceRsp;
```

where:

- `pu64ChildList` is a pointer to a list of 64-bit IEEE/MAC addresses of the child nodes in common
- `u8NumberOfChildren` is the number of child nodes in common
- `u8Status` is the status of the response

8.3 Broadcast Addresses

When sending a request using a ZDP API function, the request can be broadcast to all nodes in the network by specifying a special 16-bit network address (0xFFFF) or 64-bit IEEE/MAC address (0xFFFFFFFFFFFFFFFF). Other broadcast options are also available in order to target particular groups of nodes, as indicated in the table below.

Address Type	Broadcast Address	Target Nodes
Network (16-bit)	0xFFFF	All nodes in the network
	0xFFFD	All nodes for which ‘Rx on when idle’ is TRUE
	0xFFFC	All Routers and the Co-ordinator
IEEE/MAC (64-bit)	0xFFFFFFFFFFFFFFFF	All nodes in the network

Table 20: Broadcast Addresses and Target Nodes

9. General ZPS Resources

The chapter describes the general API resources provided in the ZigBee PRO Stack (ZPS) software.

In this chapter:

- The **ZigBee Queue** resources are detailed in Section 9.1
- The **ZigBee Timer** resources are detailed in Section 9.2
- The **Critical Section and Mutex** resources are detailed in Section 9.3

> **Note:** Amongst the general resources, functions are supplied to allow the extraction of payload data from received ZDP packets. These resources are not documented here but are provided in the header file **appZdpExtractions.h**.

9.1 ZigBee Queue Resources

The ZigBee Queue resources are concerned with creating and operating queues for passing messages from one task to another. These resources are provided in the header file **ZQueue.h**.

- The ZigBee Queue functions are described in Section 9.1.1
- The ZigBee Queue structures are described in Section 9.1.2

9.1.1 ZigBee Queue Functions

The ZigBee Queue functions are listed below, along with their page references:

ZQ_vQueueCreate

void ZQ_vQueueCreate(tszQueue **psQueueHandle*,
const uint32 *uiQueueLength*,
const uint32 *uiItemSize*,
uint8 **pu8StartQueue***);**

Description

This function creates a message queue for use by the application or stack (message queues are described in Section 5.9.1). The size of the queue and the size of a message in the queue must be specified, as well as the location in memory where the queue should start. A unique handle must also be given to the queue, where this handle is a pointer to a `tszQueue` structure that contains up-to-date information about the queue.

Parameters

psQueueHandle	Handle of message queue - this is a pointer to a `tszQueue` structure (see Section 9.1.2.1)
uiQueueLength	Size of the queue in terms of the number of messages that it can hold
uiItemSize	Size of a message in the queue, in bytes
pu8StartQueue	Pointer to the start of the message queue

Returns

None

ZQ_bQueueSend

```
bool_t ZQ_bQueueSend(void *pvQueueHandle,
                     const void *pvItemToQueue);
```

Description

This function submits a message to the specified message queue. The return code indicates whether the message was successfully added to the queue.

Parameters

pvQueueHandle	Handle of message queue
pvItemToQueue	Pointer to the message to be added to the queue

Returns

Boolean indicating the outcome of the operation:

TRUE - message successfully added to the queue

FALSE - message not added to the queue

ZQ_bQueueReceive

ZQ_bQueueReceive(void ****pvQueueHandle*,**
void ****pvItemFromQueue*);**

Description

This function obtains a message from the specified message queue. The return code indicates whether a message was successfully obtained from the queue.

Parameters

pvQueueHandle	Handle of message queue
pvItemFromQueue	Pointer to memory location to receive the obtained message

Returns

Boolean indicating the outcome of the operation:

TRUE - message successfully obtained from the queue

FALSE - message not obtained from the queue

ZQ_bQueueIsEmpty

```
bool_t ZQ_bQueueIsEmpty(void *pvQueueHandle);
```

Description

This function checks whether the specified message queue is empty. The return code indicates whether the queue is empty.

Parameters

pvQueueHandle	Handle of message queue

Returns

Boolean indicating the outcome of the operation:

TRUE - message queue is empty

FALSE - message queue is not empty

ZQ_u32QueueGetQueueSize

uint32 ZQ_u32QueueGetQueueSize(void **pvQueueHandle***);**

Description

This function obtains the capacity of the specified message queue. The return code indicates the size of the queue in terms of the number of messages that it can hold.

Parameters

pvQueueHandle	Handle of message queue

Returns

The capacity of the queue in terms of the number of messages that it can hold

ZQ_u32QueueGetQueueMessageWaiting

```
uint32 ZQ_u32QueueGetQueueMessageWaiting(
                        void *pu8QueueHandle);
```

Description

This function obtains the number of messages that are currently waiting in the specified message queue.

Parameters

pvQueueHandle	Handle of message queue

Returns

Number of messages waiting in the queue

9.1.2 ZigBee Queue Structures

9.1.2.1 tszQueue

The ZigBee queue structure `tszQueue` is shown below.

```
typedef struct
{
   uint32  u32Length;
   uint32  u32ItemSize;
   uint32  u32MessageWaiting;
   void   *pvHead;
   void   *pvWriteTo;
   void   *pvReadFrom;
}tszQueue;
```

where:

- `u32Length` is the size of the queue in terms of the number of messages that it can hold
- `u32ItemSize` is the size of a message, in bytes
- `u32MessageWaiting` is the number of messages currently in the queue
- `pvHead` is a pointer to the beginning of the queue storage area
- `pvWriteTo` is a pointer to the next free place in the storage area where a new message can be written
- `pvReadFrom` is a pointer to the next message to be read from the storage area

9.2 ZigBee Timer Resources

The ZigBee Timer functions are concerned with initialising and operating software timers. These resources are provided in the header file **ZTimer.h**.

- The ZigBee Timer functions are described in Section 9.2.1
- The ZigBee Timer structures are described in Section 9.2.2

9.2.1 ZigBee Timer Functions

The functions are listed below, along with their page references:

To use the software timers, the while loop of your application must include a call to the following function:

void ZTIMER_vTask(void);

This allows the stack software to automatically update the `ZTIMER_tsTimer` structure for each timer as the timer runs.

ZTIMER_eInit

ZTIMER_teStatus ZTIMER_eInit(ZTIMER_tsTimer **psTimers*,
uint8 *u8NumTimers***);**

Description

This function initialises a set of software timers for use by the application. A list of timers is provided in an array, in which each array element is a structure containing information on one timer (see Section 9.2.2.1). The index of an array element is used as a reference for the corresponding timer.

In order to use one of the initialised timers, it must first be opened using **ZTIMER_eOpen()**.

Parameters

psTimers	Pointer to an array of structures, where each array element contains information for one timer (see Section 9.2.2.1)
u8NumTimers	Number of timers in the above array

Returns

E_ZTIMER_OK (timers successfully initialised)

E_ZTIMER_FAIL (timers not initialised)

ZTIMER_eOpen

```
ZTIMER_teStatus ZTIMER_eOpen(
                    uint8 *pu8TimerIndex,
                    ZTIMER_tpfCallback pfCallback,
                    void *pvParams,
                    uint8 u8Flags);
```

Description

This function is used to open the specified software timer. A list of parameter values for the timer must be provided as well as a user-defined callback function that will be used to perform any operations required on expiration of the timer.

The callback function has the following prototype:

typedef void (*ZTIMER_tpfCallback)(void ***pvParam);**

where *pvParam* is a pointer to the timer parameters.

The function also includes a parameter *u8Flags* which specifies whether the timer should allow or prevent sleep. When activity checks are made to decide whether the device can enter sleep mode, the value of this flag determines if the (running) timer will stop the device from going to sleep.

Before a timer is opened, it must have been initialised in a call to **ZTIMER_eInit()**.

Parameters

pu8TimerIndex	Pointer to location containing the index number of the timer in the list of timers initialised using **ZTIMER_eInit()**
pfCallback	Pointer to the user-defined callback function for the timer
pvParams	Pointer to a list of parameter values for the timer
u8Flags	Flag indicating whether the timer should allow or prevent sleep, one of: ZTIMER_FLAG_ALLOW_SLEEP ZTIMER_FLAG_PREVENT_SLEEP

Returns

E_ZTIMER_OK (timer successfully opened)

E_ZTIMER_FAIL (timer not opened)

ZTIMER_eClose

ZTIMER_teStatus ZTIMER_eClose(uint8 *u8TimerIndex***);**

Description

This function is used to close the specified software timer when it is no longer needed. The timer must have been previously opened using **ZTIMER_eOpen()**.

Parameters

u8TimerIndex	Index number of the timer in the list of timers initialised using **ZTIMER_eInit()**

Returns

E_ZTIMER_OK (timer successfully closed)

E_ZTIMER_FAIL (timer not closed - may be running or already closed)

ZTIMER_eStart

ZTIMER_teStatus ZTIMER_eStart(uint8 *u8TimerIndex*,
uint32 *u32Time***);**

Description

This function is used to start the specified software timer. The length of time for which the timer will run must be specified in milliseconds.

Before a timer is started, it must have been opened using **ZTIMER_eOpen()**.

Once started, the timer can be stopped (before it expires) using **ZTIMER_eStop()**.

Parameters

pu8TimerIndex	Index number of the timer in the list of timers initialised using **ZTIMER_eInit()**
u32Time	The time, in milliseconds, for which the timer should run

Returns

E_ZTIMER_OK (timer successfully started)

E_ZTIMER_FAIL (timer not started)

ZTIMER_eStop

ZTIMER_teStatus ZTIMER_eStop(uint8 *u8TimerIndex***);**

Description

This function is used to stop the specified software timer (before it expires).

The timer must have been previously started using **ZTIMER_eStart()**.

Parameters

pu8TimerIndex	Index number of the timer in the list of timers initialised using **ZTIMER_eInit()**

Returns

E_ZTIMER_OK (timer successfully stopped)

E_ZTIMER_FAIL (timer not stopped - may be already stopped or expired)

ZTIMER_eGetState

ZTIMER_teState ZTIMER_eGetState(uint8 *u8TimerIndex***);**

Description

This function is used to obtain the current state of the specified software timer. The possible reported states are:

- Running
- Stopped
- Expired
- Closed

Parameters

pu8TimerIndex	Index number of the timer in the list of timers initialised using **ZTIMER_eInit()**

Returns

E_ZTIMER_STATE_CLOSED
E_ZTIMER_STATE_STOPPED
E_ZTIMER_STATE_RUNNING
E_ZTIMER_STATE_EXPIRED

9.2.2 ZigBee Timer Structures

9.2.2.1 ZTIMER_tsTimer

The ZigBee timer structure is shown below. It is used to represent a single software timer that may be used by the application.

```
typedef struct
{
    ZTIMER_teState      eState;
    uint32              u32Time;
    void               *pvParameters;
    ZTIMER_tpfCallback  pfCallback;
} ZTIMER_tsTimer;
```

where:

- `eState` represents the current state of the timer, as one of:
  - E_ZTIMER_STATE_CLOSED
  - E_ZTIMER_STATE_STOPPED
  - E_ZTIMER_STATE_RUNNING
  - E_ZTIMER_STATE_EXPIRED
- `u32Time` is the remaining time, in milliseconds, that the timer still has to run
- `pvParameters` is a pointer to a set of parameters used by the timer
- `pfCallback` is a pointer to the user-defined callback function that will be called when the timer expires - this function has the prototype:

  typedef void (*ZTIMER_tpfCallback)(void ****pvParam*);**

  where *pvParam* is a pointer to the timer parameters.

9.3 Critical Section and Mutex Resources

The Critical Section and Mutex functions are concerned with protecting sections of application code from pre-emption and re-entrancy. These resources are provided in the header file **portmacro.h**.

- The Critical Section and Mutex functions are described in Section 9.3.1
- The Critical Section and Mutex structures are described in Section 9.3.2

9.3.1 Critical Section and Mutex Functions

The functions are listed below, along with their page references:

ZPS_eEnterCriticalSection

```
uint8 ZPS_eEnterCriticalSection(
                    void *hMutex,
                    uint32* psIntStore);
```

Description

This function can be used to mark the start of a critical section of application code - this is a code section that cannot be pre-empted by an interrupt with priority level less than 12. The function **ZPS_eExitCriticalSection()** must be called at the end of the critical section.

A pointer to a 'priority level' value must be provided, which contains the current priority level of the main application thread (when critical sections are not being executed). When a critical section is entered, the priority level of the main thread is increased such that interrupts with a priority of 11 or less cannot pre-empt the main thread. At the end of the critical section, the priority level of the main thread will be returned to its value from before the critical section was entered.

Optionally, a mutex can also be applied during the critical section to protect the section from re-entrancy. If a mutex is required, a pointer must be provided to a user-defined mutex function with the following prototype:

((bool_t*) (*) (void))

This function must define and maintain a Boolean flag which indicates whether the corresponding mutex is active (TRUE) or inactive (FALSE). This flag is used by **ZPS_eEnterCriticalSection()** to determine whether the mutex is available. If this flag reads as FALSE, the mutex is applied and the above mutex function must set the flag to TRUE, but if the flag is already TRUE then the mutex cannot be applied - in the latter case, **ZPS_eEnterCriticalSection()** returns a failure.

Critical sections and mutexes are further described in Section 5.9.3.

Parameters

hMutex	Pointer to user-defined mutex function (see above) - set to NULL if no mutex is required
psIntStore	Pointer to structure containing 'priority level' value (see Section 9.3.2.1)

Returns

0x00 for success, 0x01 for failure (all other values are reserved)

ZPS_eExitCriticalSection

```
uint8 ZPS_eExitCriticalSection(
                    void *hMutex,
                    uint32* psIntStore);
```

Description

This function can be used to mark the end of a critical section of application code. The function **ZPS_eEnterCriticalSection()** must have been called at the start of the critical section.

A pointer to the ‘priority level’ value must be provided. If a mutex was used in the critical section, a pointer to the relevant mutex function must be provided in order to release the mutex.

Critical sections and mutexes are further described in Section 5.9.3.

Parameters

hMutex	Pointer to user-defined mutex function (see above) - set to NULL if no mutex was used
psIntStore	Pointer to structure containing ‘priority level’ value (see Section 9.3.2.1)

Returns

0x00 for success, 0x01 for failure (all other values are reserved)

ZPS_u8GrabMutexLock

```
uint8 ZPS_u8GrabMutexLock(
                    void *hMutex,
                    uint32* psIntStore);
```

Description

This function can be used to apply a mutex at the start of a section of application code that is to be protected from re-entrancy. The function **ZPS_u8ReleaseMutexLock()** must be called at the end of the mutex-protected section to release the mutex.

A pointer must be provided to a user-defined mutex function with the following prototype:

((bool_t*) (*) (void))

This function must define and maintain a Boolean flag which indicates whether the corresponding mutex is active (TRUE) or inactive (FALSE). This flag is used by **ZPS_u8GrabMutexLock()** to determine whether the mutex is available. If this flag reads as FALSE, the mutex is applied and the above mutex function must set the flag to TRUE, but if the flag is already TRUE then the mutex cannot be applied - in the latter case, **ZPS_u8GrabMutexLock()** returns a failure.

A pointer to a 'priority level' value must be provided, which contains the current priority level of the main application thread (when mutex protection is not being implemented). When a mutex is applied, the priority level of the main thread is increased such that interrupts with a priority of 11 or less cannot pre-empt the main thread. When the mutex is released, the priority level of the main thread will be returned to its value from before the mutex was applied.

Mutexes are further described in Section 5.9.3.2.

Parameters

hMutex	Pointer to user-defined mutex function (see above)
psIntStore	Pointer to structure containing 'priority level' value (see Section 9.3.2.1)

Returns

0x00 for success, 0x01 for failure (all other values are reserved)

ZPS_u8ReleaseMutexLock

```
uint8 ZPS_u8ReleaseMutexLock(
                        void *hMutex,
                        uint32* psIntStore);
```

Description

This function can be used to release a mutex that has been applied to a section of application code. The function **ZPS_u8GrabMutexLock()** must have been called at the start of the mutex-protected section.

A pointer to the relevant mutex function must be provided in order to release the mutex. A pointer to the 'priority level' value must also be provided.

Mutexes are further described in Section 5.9.3.2.

Parameters

hMutex	Pointer to user-defined mutex function (see above)
psIntStore	Pointer to structure containing 'priority level' value (see Section 9.3.2.1)

Returns

0x00 for success, 0x01 for failure (all other values are reserved)

9.3.2 Critical Section and Mutex Structures

9.3.2.1 u32MicroIntStorage

`u32MicroIntStorage` is a 32-bit mask of the interrupts currently enabled. The `u32MicroIntStorage` structure used in critical sections and mutex-protected sections where `u32MicroIntStorage` takes the interrupts which have been enabled.

10. Event and Status Codes

This chapter summarises the event and return/status codes of the ZigBee PRO stack.

10.1 Events

The events that can be generated by the ZigBee PRO stack are enumerated in the structure `ZPS_teAfEventType` (from the AF API), shown below.

```
typedef enum {
    ZPS_EVENT_NONE,                                    /*  0, 0x00 */
    ZPS_EVENT_APS_DATA_INDICATION,                     /*  1, 0x01 */
    ZPS_EVENT_APS_DATA_CONFIRM,                        /*  2, 0x02 */
    ZPS_EVENT_APS_DATA_ACK,                            /*  3, 0x03 */
    ZPS_EVENT_NWK_STARTED,                             /*  4, 0x04 */
    ZPS_EVENT_NWK_JOINED_AS_ROUTER,                    /*  5, 0x05 */
    ZPS_EVENT_NWK_JOINED_AS_ENDDEVICE,                 /*  6, 0x06 */
    ZPS_EVENT_NWK_FAILED_TO_START,                     /*  7, 0x07 */
    ZPS_EVENT_NWK_FAILED_TO_JOIN,                      /*  8, 0x08 */
    ZPS_EVENT_NWK_NEW_NODE_HAS_JOINED,                 /*  9, 0x09 */
    ZPS_EVENT_NWK_DISCOVERY_COMPLETE,                  /* 10, 0x0a */
    ZPS_EVENT_NWK_LEAVE_INDICATION,                    /* 11, 0x0b */
    ZPS_EVENT_NWK_LEAVE_CONFIRM,                       /* 12, 0x0c */
    ZPS_EVENT_NWK_STATUS_INDICATION,                   /* 13, 0x0d */
    ZPS_EVENT_NWK_ROUTE_DISCOVERY_CONFIRM,             /* 14, 0x0e */
    ZPS_EVENT_NWK_POLL_CONFIRM,                        /* 15, 0x0f */
    ZPS_EVENT_NWK_ED_SCAN,                             /* 16, 0x10 */
    ZPS_EVENT_ZDO_BIND,                                /* 17, 0x11 */
    ZPS_EVENT_ZDO_UNBIND,                              /* 18, 0x12 */
    ZPS_EVENT_ZDO_LINK_KEY,                            /* 19, 0x13 */
    ZPS_EVENT_BIND_REQUEST_SERVER,                     /* 20, 0x14.*/
    ZPS_EVENT_ERROR,                                   /* 21, 0x15 */
    ZPS_EVENT_APS_INTERPAN_DATA_INDICATION,            /* 22, 0x16 */
    ZPS_EVENT_APS_INTERPAN_DATA_CONFIRM,               /* 23, 0x17 */
    ZPS_EVENT_APS_ZGP_DATA_INDICATION,                 /* 24, 0x18 */
    ZPS_EVENT_APS_ZGP_DATA_CONFIRM,                    /* 25, 0x19 */
    ZPS_EVENT_TC_STATUS,                               /* 26, 0X1A */
    ZPS_EVENT_NWK_DUTYCYCLE_INDICATION,                /* 27, 0x1B */
    ZPS_EVENT_NWK_FAILED_TO_SELECT_AUX_CHANNEL,        /* 28, 0x1C */
    ZPS_EVENT_NWK_ROUTE_RECORD_INDICATION,             /* 29, 0x1D */
    ZPS_EVENT_NWK_FC_OVERFLOW_INDICATION,              /* 30, 0x1E */
    ZPS_ZCP_EVENT_FAILURE
} ZPS_teAfEventType;
```

The events in the above structure are outlined in Table 10 below.

Note: The AF structures which contain the data for the above events are detailed in Section 7.2.2.

Stack Event	Description
ZPS_EVENT_NONE	Used as initial value in structure which receives a message collected from a message queue.
ZPS_EVENT_APS_DATA_INDICATION	Indicates that data has arrived on the local node. The event provides information about the data packet through the structure `ZPS_tsAfDataIndEvent` - see Section 7.2.2.3.
ZPS_EVENT_APS_DATA_CONFIRM	Indicates whether a sent data packet has been successfully passed down the stack and has reached the next hop node towards its destination. The results are reported through the structure `ZPS_tsAfDataConfEvent` - see Section 7.2.2.4.
ZPS_EVENT_APS_DATA_ACK	Indicates that a sent message has reached its destination node. Details of the received acknowledgement are reported through the structure `ZPS_tsAfDataAckEvent` - see Section 7.2.2.5.
ZPS_EVENT_NWK_STARTED	Indicates that network has started on Co-ordinator. This is reported through the structure `ZPS_tsAfNwkFormationEvent` - see Section 7.2.2.6. ‘Permit joining’ state is set as specified in APL data structure.
ZPS_EVENT_NWK_JOINED_AS_ROUTER	Indicates that device has successfully joined network - as Router and reports allocated network address through the structure `ZPS_tsAfNwkJoinedEvent` - see Section 7.2.2.7.
ZPS_EVENT_NWK_JOINED_AS_ENDDEVICE	Indicates that device has successfully joined network as End Device and reports allocated network address through the structure `ZPS_tsAfNwkJoinedEvent` - see Section 7.2.2.7.
ZPS_EVENT_NWK_FAILED_TO_START	Indicates that network has failed to start on Co-ordinator.
ZPS_EVENT_NWK_FAILED_TO_JOIN	Indicates that device failed to join network. This is reported through the structure `ZPS_tsAfNwkJoinFailedEvent` - see Section 7.2.2.8

Table 10: Stack Events

Stack Event	Description
ZPS_EVENT_NWK_NEW_NODE_HAS_JOINED	Indicates to Co-ordinator or Router that new node has joined as child and reports details of new child through the structure `ZPS_tsAfNwkJoinIndEvent` - see Section 7.2.2.10.
ZPS_EVENT_NWK_DISCOVERY_COMPLETE	Indicates that network discovery on Router or End Device has finished and reports details of detected network through the structure `ZPS_tsAfNwkDiscoveryEvent` - see Section 7.2.2.9. This event (and associated structure) is generated for each network detected.
ZPS_EVENT_NWK_LEAVE_INDICATION	Indicates that a neighbouring node has left the network or a remote node has requested the local node to leave. Details are provided through the structure `ZPS_tsAfNwkLeaveIndEvent` - see Section 7.2.2.11.
ZPS_EVENT_NWK_LEAVE_CONFIRM	Reports the results of a node leave request issued by the local node. The results are reported through the structure `ZPS_tsAfNwkLeaveConfEvent` - see Section 7.2.2.12.
ZPS_EVENT_NWK_STATUS_INDICATION	Reports network status event from a remote or local node through the structure `ZPS_tsAfNwkStatusIndEvent` - see Section 7.2.2.13.
ZPS_EVENT_NWK_ROUTE_DISCOVERY_CONFIRM	Indicates that a route discovery has been performed. The results are reported in the structure `ZPS_tsAfNwkRouteDiscoveryConfEvent` - see Section 7.2.2.14.
ZPS_EVENT_NWK_POLL_CONFIRM	Generated on an End Device to indicate that a poll request submitted to its parent has completed. The outcome of the poll request is indicated through the structure `ZPS_tsAfPollConfEvent` - see Section 7.2.2.15.
ZPS_EVENT_NWK_ED_SCAN	Indicates that an 'energy detect' scan in the 2.4-GHz radio band has completed. The results of the scan are reported through the structure `ZPS_tsAfNwkEdScanConfEvent` - see Section 7.2.2.16.
ZPS_EVENT_ZDO_BIND	Indicates that the local node has been successfully bound to one or more remote nodes. The details of the binding are reported through the structure `ZPS_tsAfZdoBindEvent` - see Section 7.2.2.18.
ZPS_EVENT_ZDO_UNBIND	Indicates that the local node has been successfully unbound from one or more remote nodes. The details of the unbinding are reported through the structure `ZPS_tsAfZdoUnbindEvent` - see Section 7.2.2.19.
ZPS_EVENT_ZDO_LINK_KEY	Indicates that a new application link key has been received and installed, and is ready for use. The details of the link key are reported through the structure `ZPS_tsAfZdoLinkKeyEvent` - see Section 7.2.2.20.

Table 10: Stack Events

Stack Event	Description
ZPS_EVENT_BIND_REQUEST_SERVER	Indicates the results of a bound data transmission. The results are reported through the structure `ZPS_tsAfBindRequestServerEvent` - see Section 7.2.2.21.
ZPS_EVENT_ERROR	Indicates that an error has occurred on the local node. The nature of the error is reported through the structure `ZPS_tsAfErrorEvent` - see Section 7.2.2.17.
ZPS_EVENT_APS_INTERPAN_DATA_INDICATION	Indicates that an inter-PAN communication has arrived (from a node in another network). Details of the inter-PAN communication are reported through the structure `ZPS_tsAfInterPanDataIndEvent` - see Section 7.2.2.22.
ZPS_EVENT_APS_INTERPAN_DATA_CONFIRM	Indicates that an inter-PAN communication (to another network) has been sent by the local node and an acknowledgement has been received from the first hop node (this acknowledgement is not generated in the case of a broadcast). The status of the inter-PAN communication is reported through the structure `ZPS_tsAfInterPanDataConfEvent` - see Section 7.2.2.23.
ZPS_EVENT_TC_STATUS	Indicates whether the negotiation for a link key with the Trust Centre has been successful and, if so, provides the key. These details are provided through the structure `ZPS_tsAfTCstatusEvent` - see Section 7.2.2.24.
ZPS_EVENT_NWK_DUTYCYCLE_INDICATION	Relevant only for Sub Gig. Indicates the duty cycle state.
ZPS_EVENT_NWK_FAILED_TO_SELECT_AUX_CHANNEL	Relevant for Sub Gig only. Failure on a multiMAC interface to form a network on the selected channel.
ZPS_EVENT_NWK_ROUTE_RECORD_INDICATION	Indicates when a route record is received.
ZPS_EVENT_NWK_FC_OVERFLOW_INDICATION	Indicates the overflow of the frame counter when frame counter > 0x80000000.

Table 10: Stack Events

Note: Event handling is outlined in Appendix A.

10.2 Return/Status Codes

The return/status codes that can result from ZigBee PRO API function calls are divided into the following groups:

- ZDP codes - see Section 10.2.1
- APS codes - see Section 10.2.2
- NWK codes - see Section 10.2.3
- MAC codes - see Section 10.2.4
- Extended error codes - see Section 10.2.5

10.2.1 ZDP Codes

The ZDP codes are carried in request and response messages.

Name	Value	Description
ZPS_APL_ZDP_E_INV_REQUESTTYPE	0x80	The supplied request type was invalid.
ZPS_APL_ZDP_E_DEVICE_NOT_FOUND	0x81	The requested device did not exist on a device following a child descriptor request to a parent.
ZPS_APL_ZDP_E_INVALID_EP	0x82	The supplied endpoint was equal to 0x00 or between 0xF1 and 0xFF.
ZPS_APL_ZDP_E_NOT_ACTIVE	0x83	The requested endpoint is not described by a Simple descriptor.
ZPS_APL_ZDP_E_NOT_SUPPORTED	0x84	The requested optional feature is not supported on the target device.
ZPS_APL_ZDP_E_TIMEOUT	0x85	A timeout has occurred with the requested operation.
ZPS_APL_ZDP_E_NO_MATCH	0x86	The End Device bind request was unsuccessful due to a failure to match any suitable clusters.
ZPS_APL_ZDP_E_NO_ENTRY	0x88	The unbind request was unsuccessful due to the Co-ordinator or source device not having an entry in its binding table to unbind.
ZPS_APL_ZDP_E_NO_DESCRIPTOR	0x89	A child descriptor was not available following a discovery request to a parent.
ZPS_APL_ZDP_E_INSUFFICIENT_SPACE	0x8A	The device does not have storage space to support the requested operation.
ZPS_APL_ZDP_E_NOT_PERMITTED	0x8B	The device is not in the proper state to support the requested operation.
ZPS_APL_ZDP_E_TABLE_FULL	0x8C	The device does not have table space to support the operation.
ZPS_APL_ZDP_E_NOT_AUTHORIZED	0x8D	The permissions configuration table on the target indicates that the request is not authorised from this device.

Table 11: ZDP Codes

10.2.2 APS Codes

The APS codes relate to sending/receiving messages.

Name	Value	Description
ZPS_APL_APS_E_ASDU_TOO_LONG	0xA0	A transmit request failed since the ASDU is too large and fragmentation is not supported.
ZPS_APL_APS_E_DEFRAG_DEFERRED	0xA1	A received fragmented frame could not be defragmented at the current time.
ZPS_APL_APS_E_DEFRAG_UNSUPPORTED	0xA2	A received fragmented frame could not be defragmented since the device does not support fragmentation.
ZPS_APL_APS_E_ILLEGAL_REQUEST	0xA3	A parameter value was out of range.
ZPS_APL_APS_E_INVALID_BINDING	0xA4	An APSME-UNBIND.request failed due to the requested binding link not existing in the binding table.
ZPS_APL_APS_E_INVALID_GROUP	0xA5	An APSME-REMOVE-GROUP.request has been issued with a group identifier that does not appear in the group table.
ZPS_APL_APS_E_INVALID_PARAMETER	0xA6	A parameter value was invalid or out of range.
ZPS_APL_APS_E_NO_ACK	0xA7	An APSDE-DATA.request requesting acknowledged transmission failed due to no acknowledgement being received.
ZPS_APL_APS_E_NO_BOUND_DEVICE	0xA8	An APSDE-DATA.request with a destination addressing mode set to 0x00 failed due to there being no devices bound to this device.
ZPS_APL_APS_E_NO_SHORT_ADDRESS	0xA9	An APSDE-DATA.request with a destination addressing mode set to 0x03 failed due to no corresponding short address found in the address map table.
ZPS_APL_APS_E_NOT_SUPPORTED	0xAA	An APSDE-DATA.request with a destination addressing mode set to 0x00 failed due to a binding table not being supported on the device.
ZPS_APL_APS_E_SECURED_LINK_KEY	0xAB	An ASDU was received that was secured using a link key.
ZPS_APL_APS_E_SECURED_NWK_KEY	0xAC	An ASDU was received that was secured using a network key.
ZPS_APL_APS_E_SECURITY_FAIL	0xAD	An APSDE-DATA.request requesting security has resulted in an error during the corresponding security processing.

Table 12: APS Codes

Name	Value	Description
ZPS_APL_APS_E_TABLE_FULL	0xAE	An APSME-BIND.request or APSME.ADDGROUP.request issued when the binding or group tables, respectively, were full.
ZPS_APL_APS_E_UNSECURED	0xAF	An ASDU was received without any security.
ZPS_APL_APS_E_UNSUPPORTED_ATTRIBUTE	0xB0	An APSME-GET.request or APSMESET. request has been issued with an unknown attribute identifier.

Table 12: APS Codes

10.2.3 NWK Codes

The NWK codes come from the NWK layer of the stack and may be returned by any ZigBee PRO API function with a non-void return.

Name	Value	Description
ZPS_NWK_ENUM_SUCCESS	0x00	Success
ZPS_NWK_ENUM_INVALID_PARAMETER	0xC1	An invalid or out-of-range parameter has been passed
ZPS_NWK_ENUM_INVALID_REQUEST	0xC2	Request cannot be processed
ZPS_NWK_ENUM_NOT_PERMITTED	0xC3	NLME-JOIN.request not permitted
ZPS_NWK_ENUM_STARTUP_FAILURE	0xC4	NLME-NETWORK-FORMATION.request failed
ZPS_NWK_ENUM_ALREADY_PRESENT	0xC5	NLME-DIRECT-JOIN.request failure - device already present
ZPS_NWK_ENUM_SYNC_FAILURE	0xC6	NLME-SYNC.request has failed
ZPS_NWK_ENUM_NEIGHBOR_TABLE_FULL	0xC7	NLME-DIRECT-JOIN.request failure - no space in Router table
ZPS_NWK_ENUM_UNKNOWN_DEVICE	0xC8	NLME-LEAVE.request failure - device not in Neighbour table
ZPS_NWK_ENUM_UNSUPPORTED_ATTRIBUTE	0xC9	NLME-GET/SET.request unknown attribute identifier
ZPS_NWK_ENUM_NO_NETWORKS	0xCA	NLME-JOIN.request detected no networks
ZPS_NWK_ENUM_RESERVED_1	0xCB	Reserved
ZPS_NWK_ENUM_MAX_FRM_CTR	0xCC	Security processing has failed on outgoing frame due to maximum frame counter
ZPS_NWK_ENUM_NO_KEY	0xCD	Security processing has failed on outgoing frame due to no key
ZPS_NWK_ENUM_BAD_CCM_OUTPUT	0xCE	Security processing has failed on outgoing frame due CCM
ZPS_NWK_ENUM_NO_ROUTING_CAPACITY	0xCF	Attempt at route discovery has failed due to lack of table space
ZPS_NWK_ENUM_ROUTE_DISCOVERY_FAILED	0xD0	Attempt at route discovery has failed due to any reason except lack of table space
ZPS_NWK_ENUM_ROUTE_ERROR	0xD1	NLDE-DATA.request has failed due to routing failure on sending device
ZPS_NWK_ENUM_BT_TABLE_FULL	0xD2	Broadcast or broadcast-mode multicast has failed as there is no room in BTT
ZPS_NWK_ENUM_FRAME_NOT_BUFFERED	0xD3	Unicast mode multi-cast frame was discarded pending route discovery

Table 13: NWK Codes

Name	Value	Description
ZPS_NWK_ENUM_FRAME_IS_BUFFERED	0xD4	Unicast frame does not have a route available but it is buffered for automatic resend.

Table 13: NWK Codes

10.2.4 MAC Codes

The MAC codes come from the IEEE 802.15.4 MAC layer of the stack and may be returned by any ZigBee PRO API function with a non-void return. The codes are also described in the *IEEE 802.15.4 Stack User Guide (JN-UG-3024)*.

Name	Value	Description
MAC_ENUM_SUCCESS	0x00	Success
MAC_ENUM_BEACON_LOSS	0xE0	Beacon loss after synchronisation request
MAC_ENUM_CHANNEL_ACCESS_FAILURE	0xE1	CSMA/CA channel access failure
MAC_ENUM_DENIED	0xE2	GTS request denied
MAC_ENUM_DISABLE_TRX_FAILURE	0xE3	Could not disable transmit or receive
MAC_ENUM_FAILED_SECURITY_CHECK	0xE4	Incoming frame failed security check
MAC_ENUM_FRAME_TOO_LONG	0xE5	Frame too long, after security processing, to be sent
MAC_ENUM_INVALID_GTS	0xE6	GTS transmission failed
MAC_ENUM_INVALID_HANDLE	0xE7	Purge request failed to find entry in queue
MAC_ENUM_INVALID_PARAMETER	0xE8	Out-of-range parameter in function
MAC_ENUM_NO_ACK	0xE9	No acknowledgement received when expected
MAC_ENUM_NO_BEACON	0xEA	Scan failed to find any beacons
MAC_ENUM_NO_DATA	0xEB	No response data after a data request
MAC_ENUM_NO_SHORT_ADDRESS	0xEC	No allocated network (short) address for operation
MAC_ENUM_OUT_OF_CAP	0xED	Receiver-enable request could not be executed, as CAP finished
MAC_ENUM_PAN_ID_CONFLICT	0xEE	PAN ID conflict has been detected
MAC_ENUM_REALIGNMENT	0xEF	Co-ordinator realignment has been received
MAC_ENUM_TRANSACTION_EXPIRED	0xF0	Pending transaction has expired and data discarded
MAC_ENUM_TRANSACTION_OVERFLOW	0xF1	No capacity to store transaction
MAC_ENUM_TX_ACTIVE	0xF2	Receiver-enable request could not be executed, as in transmit state
MAC_ENUM_UNAVAILABLE_KEY	0xF3	Appropriate key is not available in ACL
MAC_ENUM_UNSUPPORTED_ATTRIBUTE	0xF4	PIB Set/Get on unsupported attribute

Table 14: MAC Codes

10.2.5 Extended Error Codes

If extended error handling is implemented (see Section 5.7), it provides more detail about the error that led to any one of the following function return codes:

- APS codes 0xA3 and 0xA6 (see Section 10.2.2)
- NWK code 0xC2 (see Section 10.2.3)

The extended error codes which elaborate on the above codes are provided in the `ZPS_teExtendedStatus` enumerations.

Name	Value	Description
ZPS_XS_OK	0x00	Success
ZPS_XS_E_FATAL	0x01	Fatal error - retrying will cause the error again
ZPS_XS_E_LOOPBACK_BAD_ENDPOINT	0x02	Endpoint is not valid for loopback (fatal error)
ZPS_XS_E_SIMPLE_DESCRIPTOR_NO_OUTPUT_CLUSTER	0x03	No output cluster in the Simple descriptor for this endpoint/cluster (fatal error)
ZPS_XS_E_FRAG_NEEDS_ACK	0x04	Fragmented data requests must be sent with APS ack (fatal error)
ZPS_XS_E_COMMAND_MANAGER_BAD_PARAMETER	0x05	Bad parameter has been passed to the command manager (fatal error)
ZPS_XS_E_INVALID_ADDRESS	0x06	Address parameter is out-of-range (fatal error), e.g. broadcast address when calling unicast function
ZPS_XS_E_INVALID_TX_ACK_FOR_LOCAL_EP	0x07	TX ACK bit has been set when attempting to post to a local endpoint (fatal error)
ZPS_XS_E_RESOURCE	0x08	Resource error/shortage - retrying may succeed
ZPS_XS_E_NO_FREE_NPDU	0x80	No free NPDUs (resource error) - the number of NPDUs is set in the "Number of NPDUs" property of the "PDU Manager" section of the ZPS Configuration Editor
ZPS_XS_E_NO_FREE_APDU	0x81	No free APDUs (resource error) - the number of APDUs is set in the "Instances" property of the appropriate "APDU" child of the "PDU Manager" section of the ZPS Configuration Editor
ZPS_XS_E_NO_FREE_SIM_DATA_REQ	0x82	No free simultaneous data request handles (resource error) - the number of handles is set in the "Maximum Number of Simultaneous Data Requests" field of the "APS layer configuration" section of the ZPS Configuration Editor

Table 15: Extended Error Codes

Name	Value	Description
ZPS_XS_E_NO_FREE_APS_ACK	0x83	No free APS acknowledgement handles (resource error) - the number of handles is set in the "Maximum Number of Simultaneous Data Requests with Acks" field of the "APS layer configuration" section of the ZPS Configuration Editor
ZPS_XS_E_NO_FREE_FRAG_RECORD	0x84	No free fragment record handles (resource error) - the number of handles is set in the "Maximum Number of Transmitted Simultaneous Fragmented Messages" field of the "APS layer configuration" section of the ZPS Configuration Editor
ZPS_XS_E_NO_FREE_MCPS_REQ	0x85	No free MCPS request descriptors (resource error) - there are 8 MCPS request descriptors and these are only ever likely to be exhausted under a very heavy network load or when trying to transmit too many frames too close together
ZPS_XS_E_NO_FREE_LOOPBACK	0x86	Loopback send is currently busy (resource error) - there can be only one loopback request at a time
ZPS_XS_E_NO_FREE_EXTENDED_ADDR	0x87	No free entries in the extended address table (resource error) - this table is configured in the ZPS Configuration Editor
ZPS_XS_E_SIMPLE_DESCRIPTOR_NOT_FOUND	0x88	Simple descriptor does not exist for this endpoint/cluster
ZPS_XS_E_BAD_PARAM_APSDE_REQ_RSP	0x89	Bad parameter has been found while processing an APSDE request or response
ZPS_XS_E_NO_RT_ENTRY	0x8a	No routing table entries free
ZPS_XS_E_NO_BTR	0x8b	No Broadcast transaction table entries free

Table 15: Extended Error Codes

11. ZigBee Network Parameters

This chapter lists and describes the ZigBee network parameters that can be set using the ZPS Configuration Editor described in Chapter 12.

11.1 Basic Parameters

The basic parameters are listed and described in the table below.

Parameter Name	Description	Default Value	Range
Default Extended Pan ID	The default Extended PAN ID (EPID) when adding new devices to the wireless network. The extended PAN ID is the globally unique 64-bit identifier for the network. This identifier is used to avoid PAN ID conflicts between distinct networks and must be unique among the networks overlapping in a given area. If the value is zero on the Co-ordinator, the device will use its own IEEE/MAC address as the EPID. A zero value on a Router/End Device means that the device will not look for a particular EPID when joining a network. Note that this value is the default EPID used when adding devices in the ZPS Configuration Editor. The actual EPID used for an individual device can be set via the parameter *APS Use Extended PAN ID* – see Section 11.7.	0	64 bits
Default Security Enabled	The default setting for Security Enabled when adding new devices to the wireless network.	true	true / false
Maximum Number of Nodes	The maximum number of nodes for the wireless network. This setting controls the size of tables when adding new devices to the network to ensure adequate resources are available for correct operation a network of the specified size.	20	

Table 16: ZigBee Wireless Network Parameters

The rest of the network parameters are detailed in the sections that follow, according to their area of application.

11.2 Profile Definition Parameters

Parameter Name	Description	Default Value	Range
Profile Id	The application profile identifier. This is assigned by the ZigBee Alliance for a public profile.		16 bits (Value 0 is reserved for the ZDP, 0xFFFF is wild card profile)
Name	Textual name for the profile. This is used as a prefix for generated macro definitions in **zps_gen.h**.		Valid C identifier. (“ZDP” is reserved for the ZigBee Device Profile)

Table 17: Profile Definition Parameters

11.3 Cluster Definition Parameters

Parameter Name	Description	Default Value	Range
Cluster Id	The cluster identifier. This may be defined by the ZigBee Alliance for a public cluster or may be manufacturer-specific. Clusters are designated as inputs or outputs in the simple descriptor for use in creating a binding table.		16 bits
Name	Textual name for the cluster. This is used as a prefix for generated macro definitions in **zps_gen.h**.		Valid C identifier

Table 18: Cluster Definition Parameters

11.4 Co-ordinator Parameters

Parameter Name	Description	Default Value	Range
Miscellaneous Co-ordinator Parameters			
Name	Textual name for the node. Used as a prefix when generating macro definitions in **zps_gen.h**.		Valid C identifier
Permit Joining Time	Default number of seconds for which permit joining is enabled. • 255 means permanently on • 0 means permanently off	255	0-255
Security Enabled	Specifies whether the Co-ordinator will secure communication with other devices in the network.	true	true / false
Initial Security Key	The initial key that will be used when security is enabled. These are selected from the keys available on the Trust Centre.	None	

Table 19: Co-ordinator Node Type Parameters

11.5 Router Parameters

Parameter Name	Description	Default Value	Range
Miscellaneous Router Parameters			
Name	Textual name for the node. Used as a prefix when generating macro definitions in **zps_gen.h**.		Valid C identifier
Permit Joining Time	Default number of seconds for which permit joining is enabled. • 255 means permanently on • 0 means permanently off	255	0-255
Scan Duration Time	The length of time to scan the selected RF channels when searching for a netwok to join. The time spent scanning each channel is: [aBaseSuperframeDuration x (2n + 1)] symbols where n is the value of the Scan Duration Time parameter.	3	0 – 14
Security Enabled	Specifies whether the Router will secure communication with other devices in the network.	true	true / false
Initial Security Key	The initial key that will be used when security is enabled. These are selected from the keys available on the Trust Centre.	None	

Table 20: Router Node Type Parameters

11.6 End Device Parameters

Parameter Name	Description	Default Value	Range
Miscellaneous End Device Parameters			
Name	Textual name for the node. Used as a prefix when generating macro definitions in **zps_-gen.h**.		Valid C identifier
Scan Duration Time	The length of time to scan the selected RF channels when searching for a network to join. The time spent scanning each channel is: (aBaseSuperframeDuration * (2n + 1)) symbols where n is the value of the Scan Duration Time parameter.	3	0 – 14
Security Enabled	Specifies whether the End Device will secure communication with other devices in the network.	true	true / false
Initial Security Key	The initial key that will be used when security is enabled. These are selected from the keys available on the Trust Centre.	None	
Sleeping	Indicates whether the device will turn its receiver off and enter a low-power mode. The End Device's parent will buffer any incoming data until the device returns to its normal operating state and issues a poll request.	false	true / false
Number of Poll Failures Before Rejoin	This parameter controls the number of consecutive poll failures from when the device returns to its normal operating state before attempting to find a new parent by initiating a network rejoin.	5	0 will disable this behaviour

Table 21: End Device Node Type Parameters

11.7 Advanced Device Parameters

These are advanced parameters for Co-ordinator, Router and End Device.

Parameter Name	Description	Default Value	Range
AF Parameters			
-			
AIB Parameters			
APS Designated Coordinator (read only)	Indicates that on start-up the node should assume the Co-ordinator role within the network.	true for Co-ordinator false for Routers / End Devices	true / false
APS Use Extended PAN ID	Indicates the Extended PAN ID (EPID) that the device will use. This is the globally unique 64-bit identifier for the network. This identifier is used to avoid PAN ID conflicts between distinct networks and must be unique among the networks overlapping in a given area. If the value is zero on the Co-ordinator, the device will use its own IEEE/MAC address as the EPID. A zero value on a Router/End Device means that the device will not look for a particular EPID when joining a network.	Default Extended PAN ID	64 bits
APS Inter-frame Delay	Number of milliseconds between APS data frames. Following transmission of each data block, the APS starts a timer. If there are more unacknowledged blocks to send in the current transmission window then, after a delay of *apsInterframeDelay* milliseconds the next block is passed to the NWK data service. Otherwise, the timer is set to *apscAckWaitDuration* seconds.	10	10-255

Table 22: Advanced Device Parameters

APS Max Window Size	APS fragmented data window size. Fragmentation is a way of sending messages (APDUs) longer than the payload of a single NPDU. The ASDU is segmented and split across a number of NPDUs, then reassembled at the destination. *APS Max Window Size* defines how many fragments are sent before an acknowledgement is expected. For example, if *APS Max Window Size* is set to 4 and a message is split into 16 fragments, then an acknowledgement is expected after sending fragments 1-4. Sending of fragments 5-8 does not commence until this acknowledgement is received.	8	1-8
APS Non-member Radius	Multicast non-member radius size. Defines the number of hops away from the core multi-cast members that a multi-cast transmission can be received.	2	0-7
APS Security Timeout Period	Authentication timeout period in milliseconds for nodes joining the network. If either the initiator or responder waits for an expected incoming message for a time greater than *APS Security Timeout Period* then a TIMEOUT error is generated.	1000 (6000 is advised)	
APS Use Insecure Join	Controls action when a secured network rejoin fails. If true, a join using the MAC layer association procedure is performed when a secure rejoin fails.	true	true / false
APS Layer Configuration Parameters			
APS Duplicate Table Size	The size of the APS layer duplicate rejection table. This removes duplicated APS packets.	8	1 or higher
APS Persistence Time	Time, in milliseconds, for which the resources associated with an incoming fragmented message will be retained after the complete message has been received.	100	1-255
Maximum Number of Simultaneous Data Requests	The maximum number of simultaneous APSDE data requests without APS acknowledgements. Should be set to the maximum number of target nodes in one bound transmission.	5	1 or higher

Table 22: Advanced Device Parameters

Maximum Number of Simultaneous Data Requests with Acks	The maximum number of simultaneous APSDE data requests with APS acknowledgements. Should be set to the maximum number of target nodes in one bound transmission. Note that the maximum number of APDU instances must be set to three times this value - see Table 28 on page 266.	3	1 or higher
Inter PAN	True if inter PAN functionality is enabled, see Section 5.5.1.5	false	true or false
APS Poll Period	The polling period, in milliseconds, of a sleeping End Device collecting data of any kind (received messages, received fragmented messages and all transmit acknowledgements).	100	25 or higher
Maximum Number of Received Simultaneous Fragmented Messages	Maximum number of simultaneous fragmented APSDE incoming data requests. Set to a non-zero value to enable reception of fragmented messages (note that this will increase the stack size).	0	1 or higher
Maximum Number of Transmitted Simultaneous Fragmented Messages	Maximum number of simultaneous fragmented APSDE outgoing data requests. Set to a non-zero value to enable transmission of fragmented messages (note that this will increase the stack size).	0	1 or higher
Network Layer Configuration Parameters for Co-ordinator and Routers			
Active Neighbour Table Size	Size of the active Neighbour table. Each routing node (Co-ordinator or Router) has a Neighbour table which must be large enough to accommodate entries for the node's immediate children, for its own parent and, in a Mesh network, for all peer Routers with which the node has direct radio communication.	26	1 or higher
Child Table Size	Size of the persisted sub-table of the active Neighbour table. This sub-table contains entries for the node's parent and immediate children. This value therefore determines the number of children that the node is allowed to have - it is one greater than the permitted number of children, e.g. with the default value of 5, up to 4 children are allowed. This value must not be greater than two-thirds of the *Active Neighbour Table Size* value.	5	1 or higher

Table 22: Advanced Device Parameters

Address Map Table Size	Size of the address map, which maps 64-bit IEEE addresses to 16-bit network (short) addresses. Should be set to the number of nodes in the network.	10	1 or higher
Broadcast Transaction Table Size	Size of broadcast transaction table. The broadcast transaction table stores the broadcast transaction records, which are records of the broadcast messages received by the node.	9	1 or higher
Discovery Neighbour Table Size	Size of the Discovery Neighbour table. This table keeps a list of the neighbouring devices associated with the node.	8	8-16
Route Discovery Table Size	Size of the Route Discovery table. This table is used by the node to store temporary information used during route discovery. Route Discovery table entries last only as long as the duration of a single route discovery operation.	2	1 or higher
Route Record Table Size	Size of the Route Record table. Each route record contains the destination network address, a count of the number of relay nodes to reach the destination, and a list of the network addresses of the relay nodes.	1	1 or higher
Routing Table Size	Size of the Routing table. This table stores the information required for the node to participate in the routing of message packets. Each table entry contains the destination address, the status of the route, various flags and the network address of the next hop on the way to the destination. A Routing table entry is made when a new route is initiated by the node or routed via the node. The entry is stored in the Routing table and is read whenever that route is used; the entry is only deleted if the route is no longer valid. A node is said to have routing capacity if there are free entries in the routing table.	70	1 or higher
Security Material Sets	Number of supported network keys.	2	1 or higher
Network Layer Configuration Parameters for End Devices			
Active Neighbour Table Size	Size of the active Neighbour table. Set to one (for the parent).	2	1
Address Map Table Size	Size of the address map, which maps 64-bit IEEE addresses to 16-bit network (short) addresses. Should be set to the number of nodes that the End Device application needs to communicate with plus one (for the parent).	10	1 or higher

Table 22: Advanced Device Parameters

Broadcast Transaction Table Size	Size of broadcast transaction table. The broadcast transaction table stores the broadcast transaction records, which are records of the broadcast messages received by the node.	9	1 or higher
Discovery Neighbour Table Size	Size of the Discovery Neighbour table. This table keeps a list of the neighbouring devices associated with the node.	8	8-16
Route Discovery Table Size	Not applicable - set to one.	2	1
Route Record Table Size	Not applicable - set to one.	1	1
Routing Table Size	Not applicable - set to one.	70	1
Security Material Sets	Number of supported network keys.	2	1 or higher
Stack Profile	The ZigBee Stack Profile which defines the stack features supported. Set to one for ZigBee, two for ZigBee PRO or any other value for a private stack profile.	2	0 to 15

Table 22: Advanced Device Parameters

11.7.1 Endpoint Parameters

Parameter Name	Description	Default Value	Range
Application Device Id	Device ID for the endpoint.		
Application Device Version	Version number for the device.		
Enabled	Whether the endpoint is enabled or disabled.	true	true / false
End Point Id	The endpoint number (must be unique within the network).		1-240
Name	Textual name for the endpoint. Used as a prefix when generating macro definitions.		Valid C identifier
Profile	The application profile for the endpoint. This as a link to a profile definition.		

Table 23: Endpoint Parameters

Input Cluster

Specifies that the endpoint will receive for the specified cluster.

Parameter Name	Description	Default Value	Range
Cluster	A link to an input cluster that will receive on the endpoint.		
Receive APDU	A link to an APDU that will buffer any incoming messages.		
Discoverable	Defines whether the input cluster will be present in the endpoints simple descriptor which is used for service discovery.	true	true / false

Table 24: Input Cluster Parameters

Output Cluster

Specifies that the endpoint will transmit for the specified cluster.

Parameter Name	Description	Default Value	Range
Cluster	A link to an output cluster that will transmit on the endpoint.		
Transmit APDUs	List of APDUs that will be used to transmit the cluster.		
Discoverable	Defines whether the input cluster will be present in the endpoints Simple descriptor which is used for service discovery.	true	true / false

Table 25: Output Cluster Parameters

11.7.2 Bound Addressing Table

Specifies that the device should include a Binding table. Binding is optional. If Binding tables are used, they are located on any node which is a source for a binding, but the ZigBee Co-ordinator handles end device bind requests on behalf of all devices in the network. Nodes that use Binding tables should be allocated enough Binding table entries to handle their own communication needs.

Parameter Name	Description	Default Value	Range
Size	The size of the Binding table. Each binding table entry contains: • The node address and endpoint number of the source of the binding • The node address and endpoint number of the destination of the binding • The cluster ID for the binding If a binding is one-to-many then a table entry is required for each destination.		

Table 26: Bound Addressing Table Parameters

11.7.3 PDU Manager

The Protocol Data Unit Manager (PDUM) configuration is mandatory and must always be present.

Parameter Name	Description	Default Value	Range
Number of NPDUs	The number of NPDUs available to the ZigBee stack. These are internal to the stack.	16	8 or higher

Table 27: PDU Manager Parameters

APDU

Specifies a buffer to contain instances of a cluster.

Parameter Name	Description	Default Value	Range
Instances	The maximum number of instances of this APDU. Note that this value must be set to three times the value of the parameter *Maximum Number of Simultaneous Data Requests with Acks* - see Table 22 on page 260.		
Name	The name of the APDU. This is the identifier that should be used in the application C code to refer to the APDU.		Valid C identifier
Size	The maximum size of the APDU.		

Table 28: APDU Parameters

11.7.4 Group Addressing Table

Specifies that the device contains a Group table.

Parameter Name	Description	Default Value	Range
Size	The size of the Group table. Group membership for endpoints on the current device is controlled by adding and removing entries in the Group table.		

Table 29: Group Addressing Table Parameters

11.7.5 RF Channels

Specifies the default RF channels that the device will operate on. If not present, the default will be all channels.

Parameter Name	Description	Default Value	Range
Channel x (x=11-26)	Control for channel x – setting to true includes the channel in energy scan. By default, only channel 15 is included.	true for x=15, false for all other values	true / false

Table 30: RF Channels Parameters

11.7.6 MAC Interface Table

Specifies the number of interfaces which are present on this device. One interface must be setup.

Parameter Name	Description	Default Value	Range
Channel List size (Needs to be set to 1 if only 2.4g)	Number of channel masks	1	No value ranges
Enabled	Is interface enabled/disabled	Enabled	TRUE/FALSE
Index	Index of the entry	0	0 to channel list size
Radio Type	2.4G or SubGig	RT2400MHz	RT2400MHz/ RT868MHz/ RT915MHz
Routers Allowed	Are routers allowed on this interface	FALSE	TRUE/FALSE

Table 31: MAC Interface Table

11.7.7 Node Descriptor

This is mandatory and defines the type and capabilities of the node.

Parameter Name	Description	Default Value	Range

Table 32: Node Descriptor Parameters

Descriptor Availability Parameters			
Complex Descriptor Available	Complex descriptors are not supported. Not editable.	false	false
User Descriptor Available	Indicates whether a user descriptor is present. Not editable.	false	true / false
Descriptor Capabilities Parameters			
Extended Active Endpoint List Available	Indicates whether an extended active endpoint list is available. Not editable.	false	false
Extended Simple Descriptor List Available	Indicates whether an extended simple descriptor list is available. Not editable.	false	false
MAC Capability Flags			
Allocate Address	Indicates whether the device will allocate short (network) addresses or not. Not editable.		true / false
Alternate PAN Coordinator	Indicates whether the device will act as an alternative PAN Co-ordinator. Not editable.		true / false
Device type	Indicates whether the device is a Full Functionality Device (FFD) or Reduced Functionality Device (RFD). Not editable.		true / false
Power source	Indicates whether the device is mains powered or not. Not editable.		true / false
Rx On When Idle	Indicates whether the device has its receiver enabled while the device is idle. Not editable.		true / false
Security	Indicates whether the device uses high or standard security. Only standard security is supported. Not editable.	false	true / false
Miscellaneous parameters			
APS flags	Not editable.	0	0
Frequency Band	Frequency band of radio. The the JN518x and K32W041/K32W061 devices only support the 2.4 GHz band. Not editable.	2.4 GHz	2.4 GHz
Logical Type	The device type: Co-ordinator, Router, or End Device. Not editable.		ZC/ZR/ZED
Manufacturer Code	The manufacturer ID code. These are allocated by the ZigBee Alliance.		0 - 65535
Maximum buffer size	The maximum buffer size. Not editable.	127	

Table 32: Node Descriptor Parameters

Maximum incoming transfer size	The maximum incoming transfer size supported by the device. This is calculated from the APDU sizes for input clusters. Not editable.		
Maximum outgoing transfer size	The maximum incoming transfer size supported by the device. This is calculated from the APDU sizes for output clusters. Not editable		
System Server Capabilities parameters			
Backup binding table cache	Indicates if the node can act as a backup binding table cache. Not supported and not editable.	false	true / false
Backup discovery cache	Indicates if the node can act as a backup discovery cache. Not supported and not editable.	false	true / false
Backup trust center	Indicates if the node can act as a backup trust centre. Not supported and not editable.	false	true / false
Network manager	Indicates if the node can act as a network manager. Not editable.	false	true / false
Primary binding table cache	Indicates if the node can act as a primary binding table cache. Not supported and not editable.	false	true / false
Primary discovery cache	Indicates if the node can act as a primary discovery cache. Not supported and not editable.	false	true / false
Primary trust center	Indicates if the node can act as a trust center. Not editable.	false	true / false

Table 32: Node Descriptor Parameters

11.7.8 Node Power Descriptor

The Node Power descriptor for the device is mandatory.

Parameter Name	Description	Default Value	Range
Available Power Sources parameters			
Constant power	Indicates whether a constant power source is available.	false	true / false
Disposable Battery	Indicates whether a disposable battery power source is available.	false	true / false
Rechargable Battery	Indicates whether a rechargable battery power source is available.	false	true / false
Miscellaneous parameters			
Default power mode	The default power mode of the device.	Synchronised with RxOn-WhenIdle	Synchronised with RxOnWhenIdle / Peri-odic / Constant Power
Default power source	The default power source of the device.	Constant / rechargeable / disposable	Constant

Table 33: Node Power Descriptor Parameters

11.7.9 Key Descriptor Table

Specifies that the device should contain a Key Descriptor Table (for APS security).

Parameter Name	Description	Default Value	Range
Size	The size of the key descriptor table.		1 or higher

Table 34: Key Descriptor Table Parameters

Preconfigured Key

Specifies a pre-configured link key for the Key Descriptor Table.

Parameter Name	Description	Default Value	Range
IEEE address	The IEEE address to use with the key.		64 bit
Key	The pre-configured key value.		128 bit

Table 35: Preconfigured Key Parameters

11.7.10 Trust Centre

Specifies that the device will have the capability to act as a Trust Centre.

Parameter Name	Description	Default Value	Range
Device Table Size	The size of the Trust Centre's device table.	Maximum Number of Nodes setting from the ZigBee PRO Wireless Network	1 or higher

Table 36: Trust Centre Parameters

A Trust Centre link key must be pre-set in the Key Descriptor Table (see Section 11.7.9) on each node.

11.7.11 ZDO Configuration

Specifies which ZigBee Device Object (ZDO) servers are present on the device. Most of these are mandatory for a ZCP.

The ZDO configuration parameters are detailed in the following categories:

Default Server

Mandatory. Replies to any unimplemented server requests.

Parameter Name	Description	Default Value	Range
Output APDU	The APDU to use when replying to unimplemented server request messages.	apduZDP	

Table 37: Default Server Parameters

ZDO Client

Mandatory. Processes ZDO client messages.

Parameter Name	Description	Default Value	Range
Output APDU	The APDU to use when replying to ZDO client messages.	apduZDP	

Table 38: ZDO Client Parameters

Device Annce Server

Mandatory. Processes device announcements.

Parameter Name	Description	Default Value	Range
Output APDU	The APDU to use when replying to device announcement messages.	apduZDP	

Table 39: Default Server Parameters

Active Ep Server

Mandatory. Processes active endpoint requests.

Parameter Name	Description	Default Value	Range
Output APDU	The APDU to use when replying to active endpoint request messages.	apduZDP	

Table 40: Active Ep Server Parameters

Nwk Addr Server

Mandatory. Processes network address discovery requests.

Parameter Name	Description	Default Value	Range
Output APDU	The APDU to use when replying to network address discovery request messages.	apduZDP	

Table 41: Nwk Addr Server Parameters

IEEE Address Server

Mandatory. Processes IEEE address discovery requests.

Parameter Name	Description	Default Value	Range
Output APDU	The APDU to use when replying to IEEE address discovery request messages.	apduZDP	

Table 42: IEEE Address Server Parameters

System Server Discovery Server

Mandatory. Processes system server discovery requests.

Parameter Name	Description	Default Value	Range
Output APDU	The APDU to use when replying to system server discovery request messages.	apduZDP	

Table 43: System Server Discovery Server Parameters

Permit Joining Server

Mandatory. Processes 'permit joining' requests.

Parameter Name	Description	Default Value	Range
Output APDU	The APDU to use when replying to permit joining request messages.	apduZDP	

Table 44: Permit Joining Server Parameters

Node Descriptor Server

Mandatory. Processes Node descriptor requests.

Parameter Name	Description	Default Value	Range
Output APDU	The APDU to use when replying to node descriptor request messages.	apduZDP	

Table 45: Node Descriptor Server Parameters

Power Descriptor Server

Mandatory. Processes Node Power descriptor requests.

Parameter Name	Description	Default Value	Range
Output APDU	The APDU to use when replying to power descriptor request messages.	apduZDP	

Table 46: Power Descriptor Server Parameters

Match Descriptor Server

Mandatory. Processes Match descriptor requests.

Parameter Name	Description	Default Value	Range
Output APDU	The APDU to use when replying to Match descriptor request messages.	apduZDP	

Table 47: Match Descriptor Server Parameters

Simple Descriptor Server

Mandatory. Processes simple descriptor requests.

Parameter Name	Description	Default Value	Range
Output APDU	The APDU to use when replying to Simple descriptor request messages.	apduZDP	

Table 48: Simple Descriptor Server Parameters

Mgmt Lqi Server

Mandatory. Processes management LQI requests.

Parameter Name	Description	Default Value	Range
Output APDU	The APDU to use when replying to Link Quality Indicator (LQI) request messages.	apduZDP	

Table 49: Mgmt Lqi Server Parameters

Mgmt Rtg Server

Mandatory. Processes management routing requests.

Parameter Name	Description	Default Value	Range
Output APDU	The APDU to use when replying to management routing request messages.	apduZDP	

Table 50: Mgmt Rtg Server Parameters

Mgmt Leave Server

Mandatory. Processes management leave requests.

Parameter Name	Description	Default Value	Range
Output APDU	The APDU to use when replying to management leave request messages.	apduZDP	

Table 51: Mgmt Leave Server Parameters

Mgmt NWK Update Server

Mandatory. Processes management network update requests.

Parameter Name	Description	Default Value	Range
Output APDU	The APDU to use when replying to management network update request messages.	apduZDP	

Table 52: Mgmt NWK Update Server Parameters

Bind Unbind Server

Mandatory. Processes both bind and unbind requests.

Parameter Name	Description	Default Value	Range
Output APDU	The APDU to use when replying to bind and unbind request messages.	apduZDP	

Table 53: Bind Unbind Server Parameters

Extended Active Ep Server

Mandatory. Processes extended active endpoint discovery requests.

Parameter Name	Description	Default Value	Range
Output APDU	The APDU to use when replying to extended active endpoint discovery request messages.	apduZDP	

Table 54: Active Ep Server Parameters

Extended Simple Descriptor Server

Mandatory. Processes extended Simple descriptor discovery requests.

Parameter Name	Description	Default Value	Range
Output APDU	The APDU to use when replying to extended Simple descriptor discovery request messages.	apduZDP	

Table 55: Extended Simple Descriptor Server Parameters

End Device Bind Server

Mandatory (Co-ordinator only). Processes End Device bind requests.

Parameter Name	Description	Default Value	Range
Output APDU	The APDU to use when replying to end device bind request messages.	apduZDP	
Timeout	Number of seconds before timing out an End Device bind request.	5	1 or higher
Bind Num Retries	Number of binding retries attempted if a binding request (zdo_bind_req or end_device_bind_req) fails.		

Table 56: End Device Bind Server Parameters

Parent Announcement Server

Mandatory on the co-ordinator and router devices.

Parameter Name	Description	Default Value	Range
Output APDU	The APDU to use when replying to parent announce broadcast messages.		

Table 57: End Parent Announcement Server

Management Enhanced Network Update Server

Mandatory for Sub Gig; optional for 2.4G.

Parameter Name	Description	Default Value	Range
Output APDU	The APDU to use when replying to management enhanced network update request messages.	apduZDP	

Table 58: Management Enhanced Network Update Server

MIB IEEE List Server

Mandatory on routers in Sub Gig; optional on 2.4G.

Parameter Name	Description	Default Value	Range
Output APDU	The APDU to use when replying to MIB IEEE request messages.	apduZDP	

Table 59: MIB IEEE List Server

Part III:
Network Configuration

12. ZPS Configuration Editor

In developing a ZigBee PRO application, certain static configuration is required before the application is built. The ZPS Configuration Editor is used to simplify this configuration. This editor is supplied as an NXP plug-in for Eclipse and is provided in the ZigBee 3.0 SDK (see Section 4.1.2). The plug-in is suitable for use with MCUXpresso.

The ZPS Configuration Editor provides a convenient way to set ZigBee network parameters, such as the properties of the Co-ordinator, Routers and End Devices (for example, by setting elements of the device descriptors). This chapter describes how to use the ZPS Configuration Editor, as follows:

- Section 12.1 describes how the ZigBee network configuration is used in the application build process
- Section 12.2 describes how to access the ZPS Configuration Editor wizard
- Section 12.3 provides an overview of the interface provided by the ZPS Configuration Editor
- Section 12.4 describes how to use the ZPS Configuration Editor to perform important configuration tasks

12.1 Configuration Principles

The build process for a ZigBee PRO application takes a number of configuration files, in addition to the application source file and header file. The following files are generated from the MCUXpresso IDE to feed into the build process:

- ZigBee PRO Stack files:
  - **zps_gen.c**
  - **zps_gen.h**
- PDUM files:
  - **pdum_gen.c**
  - **pdum_gen.h**
- Other files:
  - **port.c**
  - **portasm.h**
  - **portmacro**
  - **irq.s**

All of the above files are produced according to the same basic principle. The ZPS Configuration Editor in the MCUXpresso IDE is used to edit the configuration data and output this data as an XML file (the XML file can be coded manually, outside of MCUXpresso, but this is not recommended). As part of the build process, the application's makefile invokes command line utilities that use the XML files to generate the files listed above.

The full build process is illustrated in Figure 12.

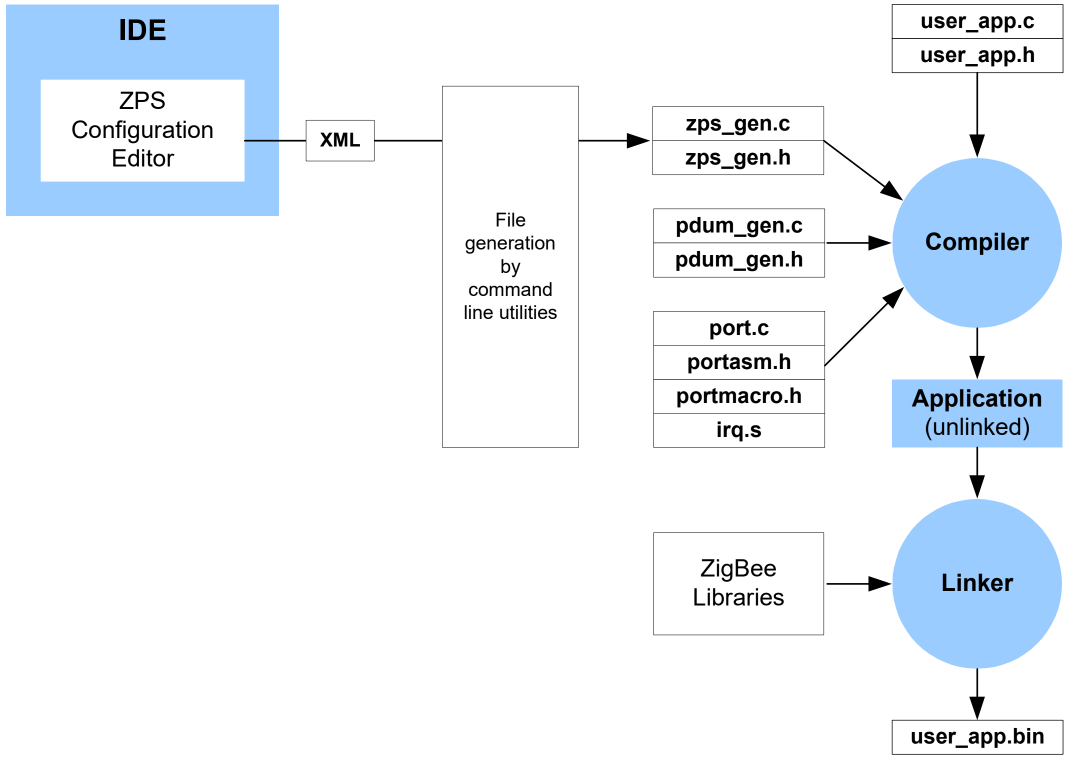


Figure 12: Application Build Process

12.2 ZPS Configuration Editor Wizard

Before you can start to create a new ZigBee PRO stack configuration, the ZPS Configuration Editor plug-in must be installed in MCUXpresso.

To check if the plug-in is already installed, start MCUXpresso and select **File > New > Other** from the main menu. Check that a **Jennic** option exists in the **Select a Wizard** dialogue box - expanding the **Jennic** option should show "ZBPro Configuration", as illustrated in the screenshot below.

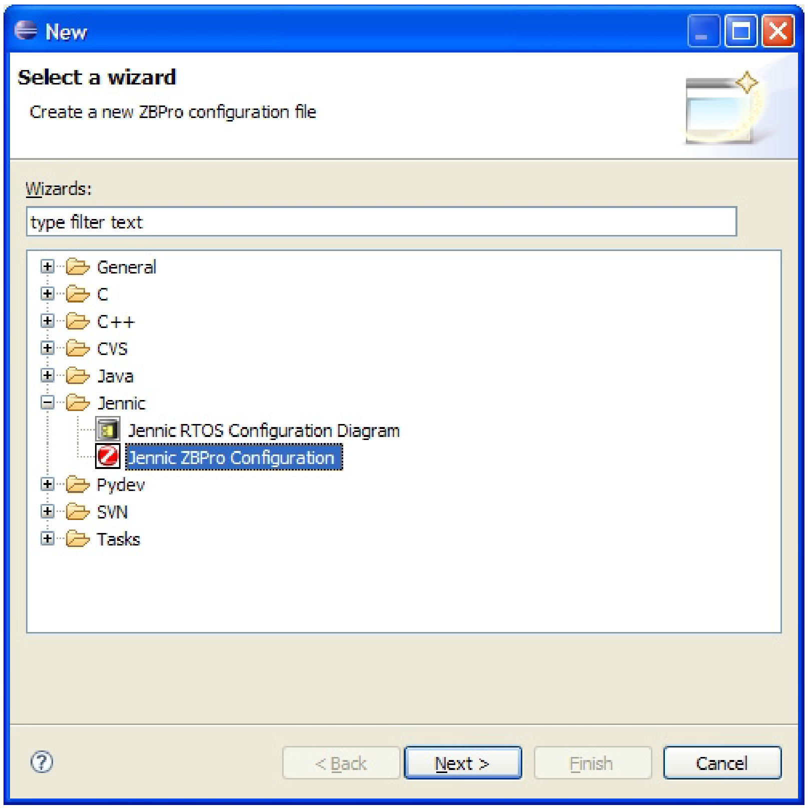


Figure 13: Select a Wizard

Using the wizard highlighted in the screenshot above, you can start to create a new ZigBee PRO configuration, as described in Section 12.4.1.

If the wizard is not present, install the ZPS Configuration Editor plug-in, which is supplied in the ZigBee 3.0 SDK.

12.3 Overview of ZPS Configuration Editor Interface

The ZPS Configuration Editor allows ZigBee network parameters to be configured through an easy-to-use Windows Explorer-style interface. This interface is outlined below.

The parameter values for the whole network are stored in a file with extension **.zpscfg**, and the ZPS Configuration Editor provides a convenient way to view and edit the contents of this file. The network parameters are presented in an expandible tree, as shown below.

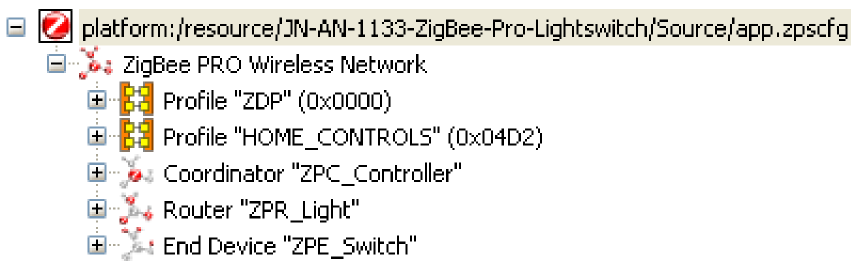


Figure 14: Network Parameters

Entries that sit at the same level in the tree are termed ‘siblings’, while an entry that sits under another entry in the tree (a sub-entry) is termed a ‘child’.

The top level of the tree shows the Extended PAN ID. The next level shows the following siblings:

- Entries for the ZigBee application profiles used in the network
- Entry for the Co-ordinator
- Entries for the Routers
- Entries for the End Devices

The information under each of these entries is described below.

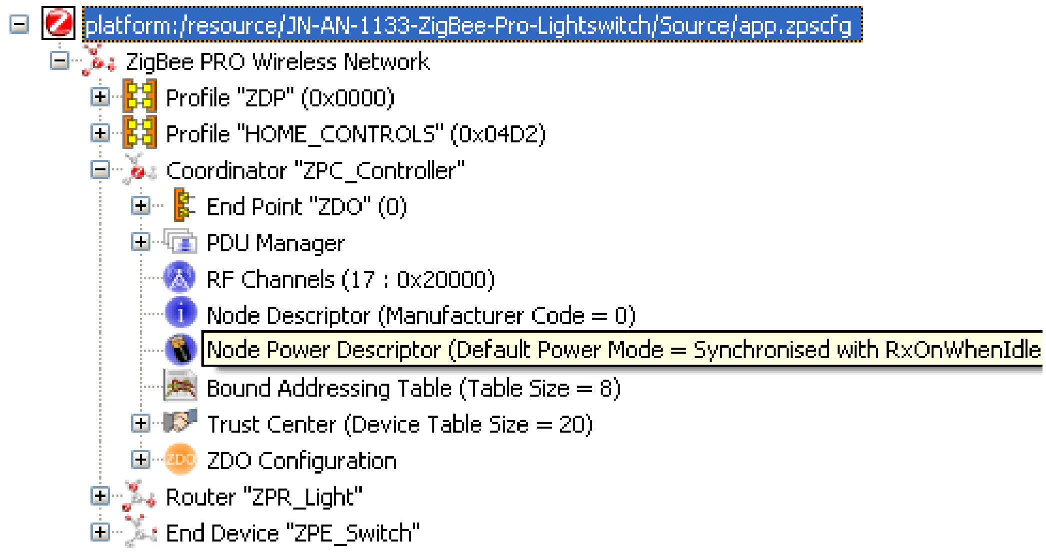
Profile

An application profile has a numeric ID and a name. The Profile entry contains child entries for the clusters supported by the profile - each cluster is identified by a numeric ID and a name.

> **Note:** There must be entries for all application profiles supported by the network. An individual device may not use all profiles, although a device can use more than one profile to support multiple features (for example, measurement of temperature, humidity and light level).

Co-ordinator

The Co-ordinator entry contains a name and a number of associated parameters, mainly related to the APS and NWK layers of the ZigBee PRO stack.



The child entries for the Co-ordinator are shown above and include the following:

- Endpoint entries, one for each endpoint on the Co-ordinator, with each endpoint having child entries specifying the input and output clusters used (note that each input cluster must be paired with an APDU)
- PDU Manager, with child entries specifying the APDUs used
- Channel Mask, specifying the 2.4-GHz band channels to scan when creating the network
- Node Descriptor for the Co-ordinator
- Node Power Descriptor for the Co-ordinator

Router

Each Router entry contains a name and a number of associated parameters, mainly related to the APS and NWK layers of the ZigBee PRO stack. The child entries for a Router include the following:

- Endpoint entries, one for each endpoint on the Router, with each endpoint having child entries specifying input and output clusters used (note that each input cluster must be paired with an APDU)
- PDU Manager, with child entries specifying the APDUs used
- Channel Mask, specifying the 2.4-GHz band channels to scan when attempting to join a network
- Node Descriptor for the Router
- Node Power Descriptor for the Router

End Device

Each End Device entry contains a name and a number of associated parameters, mainly related to the APS and NWK layers of the ZigBee PRO stack. The child entries for an End Device include the following:

- Endpoint entries, one for each endpoint on the End Device, with each endpoint having child entries specifying the input and output clusters used (note that each input cluster must be paired with an APDU)
- PDU Manager, with child entries specifying the APDUs used
- Channel Mask, specifying the 2.4-GHz band channels to scan when attempting to join a network
- Node Descriptor for the End Device
- Node Power Descriptor for the End Device

12.4 Using the ZPS Configuration Editor

Note: This section assumes that you wish to add a ZigBee PRO stack configuration to a project which you have already created in MCUXpresso (in this example, HelloWorld).

12.4.1 Creating a New ZPS Configuration

Step 1 In MCUXpresso, start the ZPS Configuration Editor wizard. To do this, follow the menu path **File > New > Other** and in the **Select a Wizard** dialogue box, select "Jennic ZBPro Configuration" and click **Next** (shown in Figure 13 on page 283).

The **New** dialogue box opens for the ZBPro Configuration.

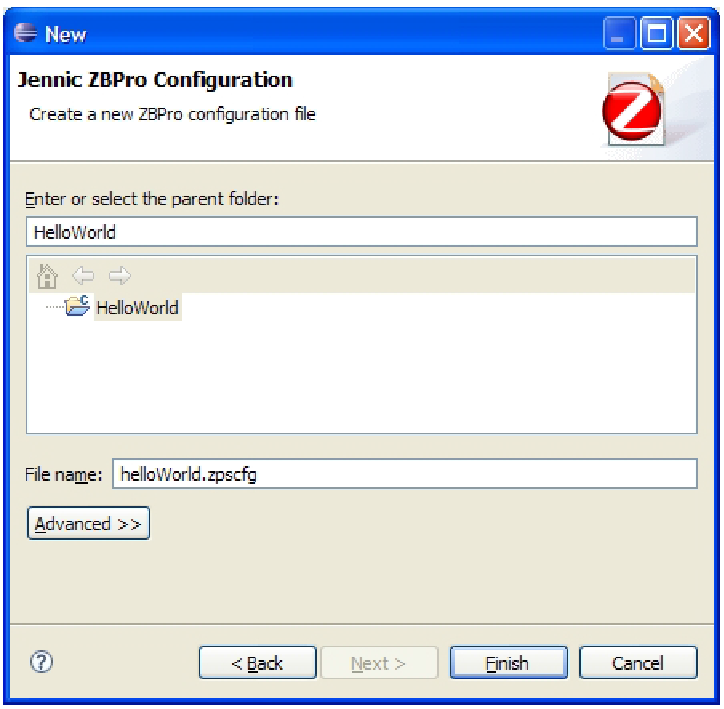


Figure 15: New ZPS Configuration

Step 2 Click on your project to select it as the parent folder. In the **File name** field, enter a name for the configuration file (keep the extension **.zpscfg**) and then click **Finish**.

A new configuration (with the default set of parameters) will open in the editor, as shown below.

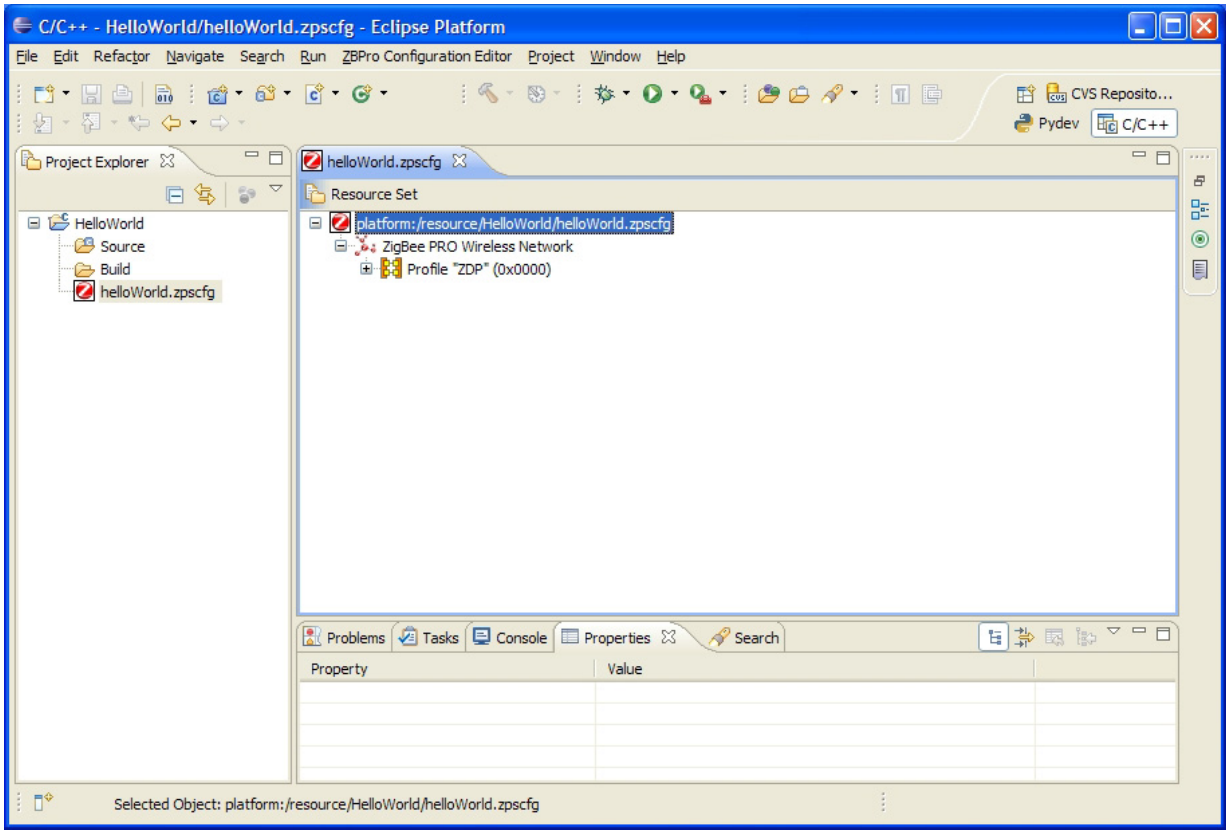


Figure 16: ZPS Configuration Editor Window

12.4.2 Adding Device Types

To add devices

Step 1 Right-click on **ZigBee PRO Wireless Network** and select **New Child > Coordinator** from the drop-down menu. This inserts a Co-ordinator with the minimum necessary child elements.

Step 2 Add Routers and End Devices in the same way, as required. The network can only have one Co-ordinator, but as many different Router or End Device types (i.e. running different application features and with different endpoints) as required.

Step 3 For each new device, use the **Properties** tab (bottom pane) to enter the required top-level parameters. For a sleeping End Device, set **Sleeping** to True (by right-clicking on the value and using the drop-down box).

> **Note:** To display the advanced properties, click the **Advanced** tool button to the right of the **Properties** view tab - see Section 12.4.4. These properties are all set to default values and can be left unchanged, unless specific changes are required.

To add a profile

Step 1 Right-click on **ZigBee PRO Wireless Network** and select **New Child > Profile** from the drop-down menu. This inserts a profile with no child elements.

Step 2 Edit the properties in the **Properties** tab to set **Name** and **Id** for the new profile.

To add clusters to the new profile

Step 1 Right-click on the new profile created above and select **New Child > Cluster** from the drop-down menu.

Step 2 Edit the properties in the **Properties** tab to set **Name** and **Id** for the new cluster.

Step 3 Repeat Step 1 and Step 2 to add more clusters, as required.

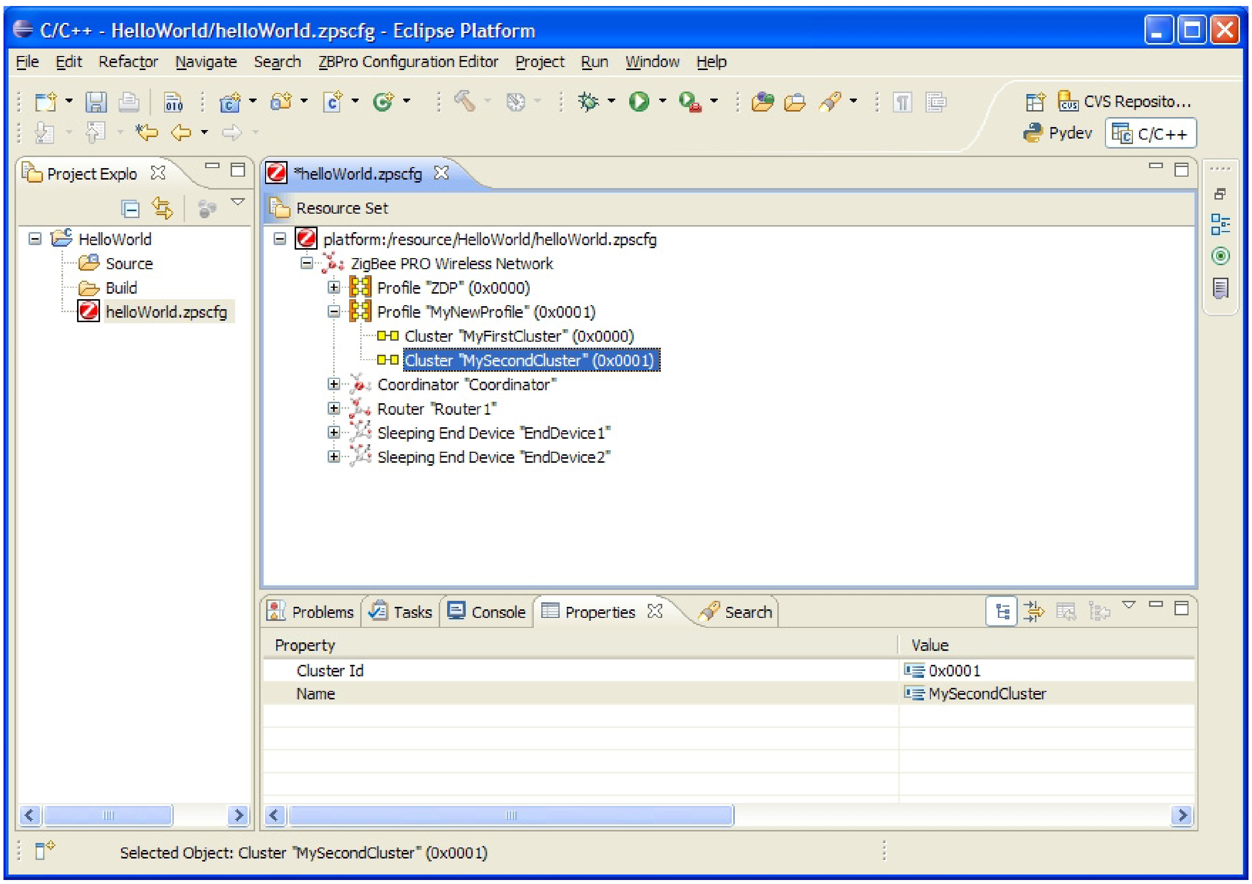


Figure 17: Cluster Properties

12.4.3 Setting Co-ordinator Properties

To set the channel mask and Node Power descriptor

Step 1 Expand the Co-ordinator node in the editor. This will reveal the default set of features for the Co-ordinator, ZDO endpoint and ZDO servers.

Step 2 Click on the **RF Channels** element to modify the channel mask.

There are 16 channels available, numbered 11 to 26, which are now shown in the **Properties** tab. A single channel or a set of channels can be selected for the channel mask, as required.

Step 3 In the **Properties** tab, set the desired channel(s) to true (by right-clicking on the value and using the drop-down box).

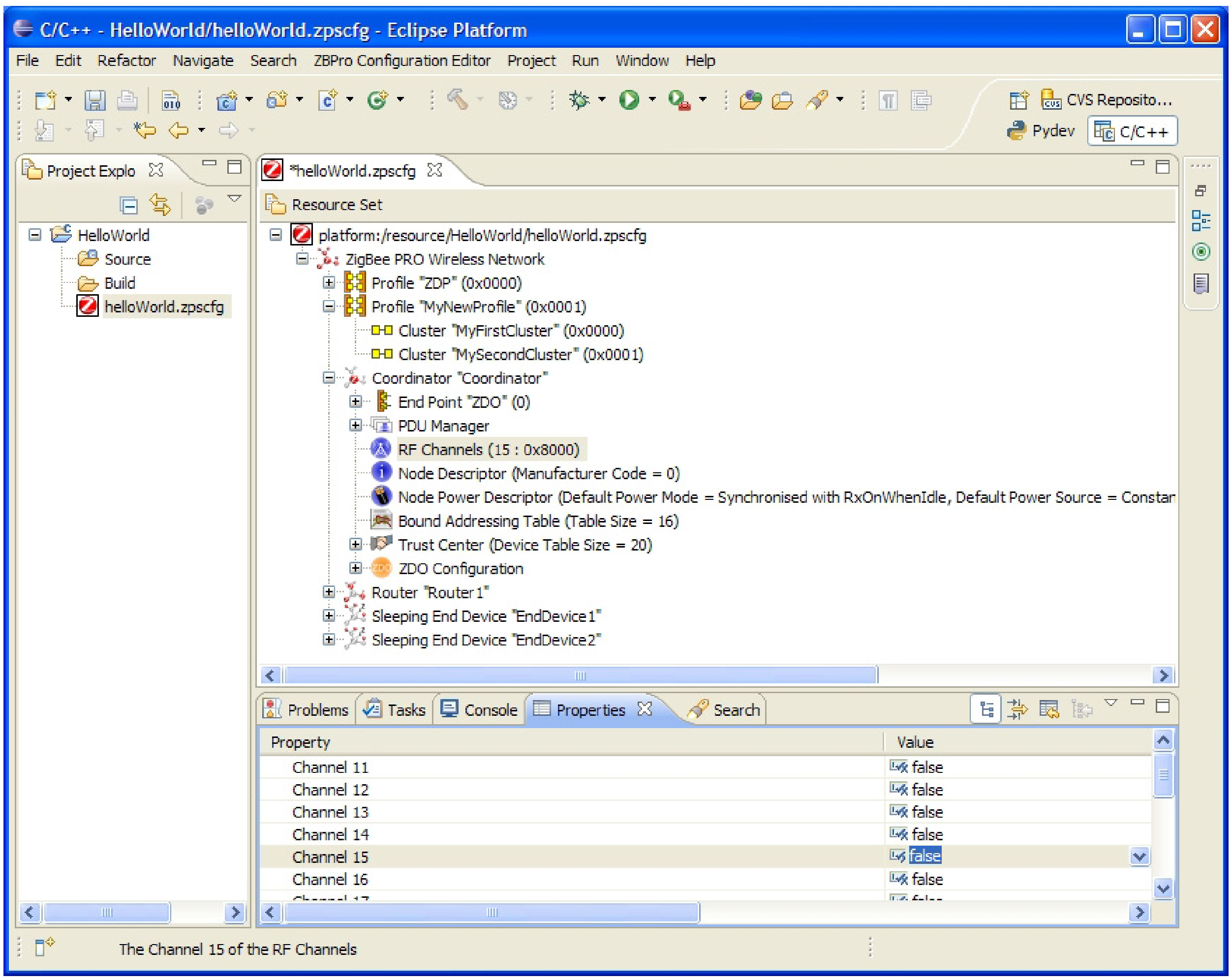


Figure 18: Channel Mask Selection

Step 4 Click to select the **Node Power Descriptor**.

Step 5 Edit the properties in the **Properties** tab, as required.

To add a new endpoint

Step 1 Right-click on the Co-ordinator node and select **New Child > End Point** from the drop-down menu.

Step 2 Edit the properties in the **Properties** tab to set **Name** and **Profile** for the endpoint (the profile is selected from the drop-down box).

Step 3 Repeat Steps 1 and 2 for as many endpoints as are required.

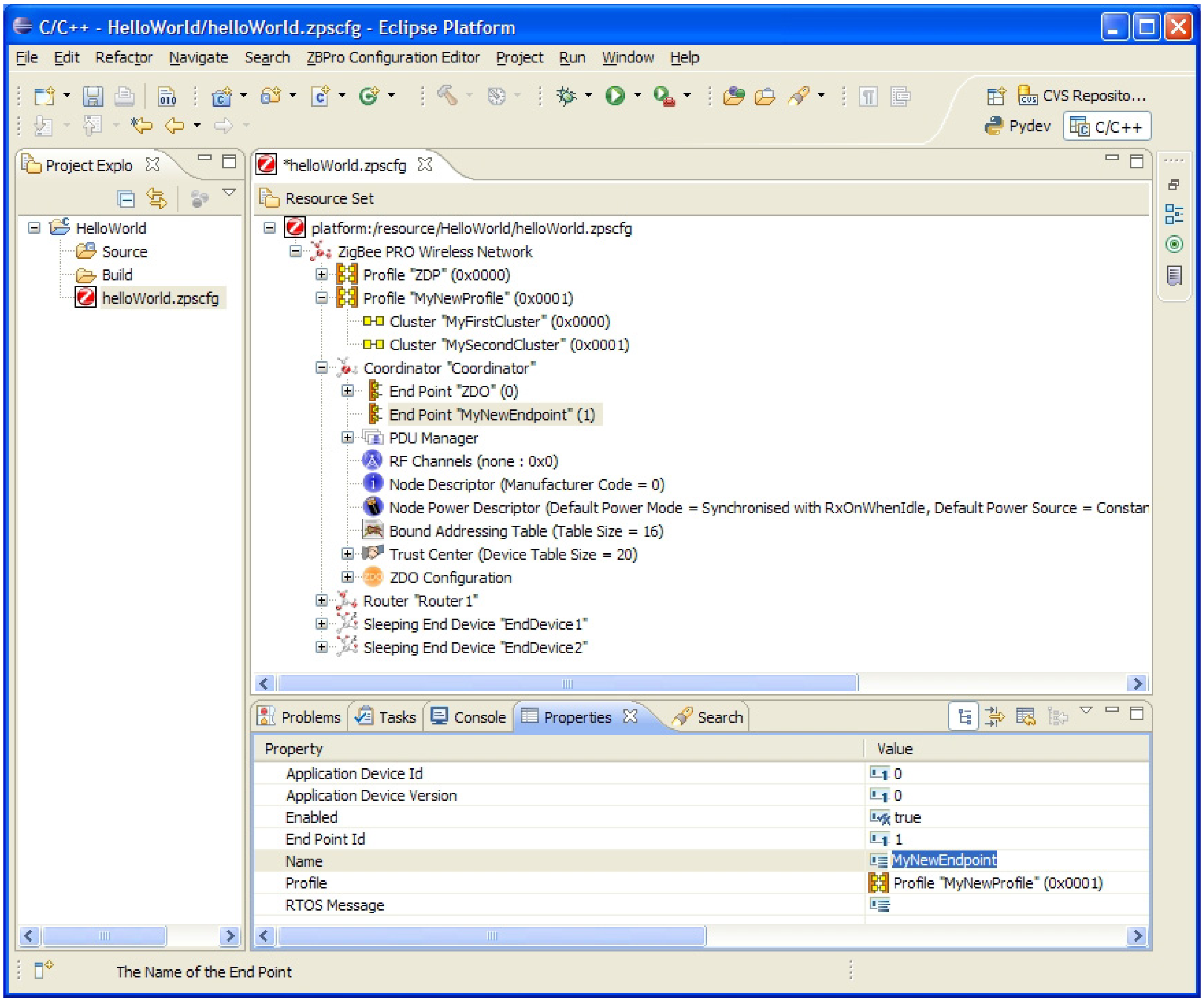


Figure 19: Endpoint Properties

To add an APDU

At least one APDU is required before an endpoint can send or receive data. The same APDU can be used to send and receive data, or different APDUs can be set up for send and receive - this allows control of buffering and memory resources, and is the decision of the application designer.

Step 1 Right-click on **PDU Manager** and select **New Child > APDU** from the drop-down menu.

Step 2 Edit the properties in the **Properties** tab to set **Name**, **Instances** (number of) and **Size** (of each instance - this should be set to the size of the largest APDU to be received).

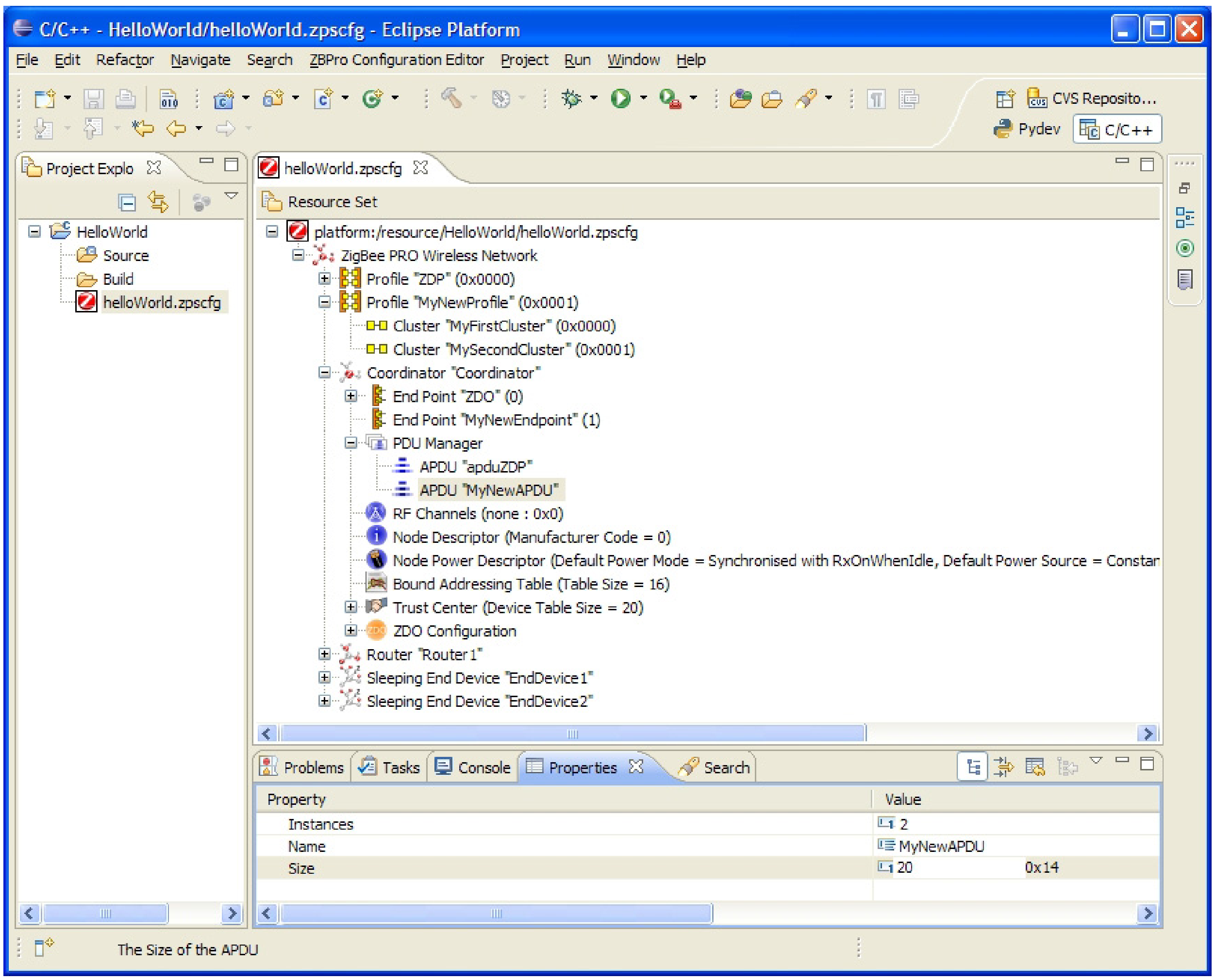


Figure 20: APDU Properties

To add input and output clusters to an endpoint

Step 1 Right-click on the endpoint and select **New Child > Input Cluster** or **New Child > Output Cluster**, as required, from the drop-down menu.

Step 2 Edit the properties in the **Properties** tab to set **Cluster** - select from the available clusters in the drop-down list.

Step 3 Edit the **Rx APDU** or **Tx APDU** property to assign an APDU to the cluster - select from the available APDUs in the drop-down list.

To receive data, a cluster must have an assigned APDU. The same cluster can be both an input and output cluster, i.e. it will both send and receive data.

When an endpoint with an output cluster sends data, the receiving endpoint must have an input cluster in order to receive the data, otherwise the stack will reject it and will not notify the receiving endpoint. However, the Default cluster can be added to the endpoint in order to deal with received data that is destined for input clusters not supported by the endpoint (see the Note below this procedure).

Step 4 Repeat Step 1 to Step 3 to add as many clusters as are required for the endpoint.

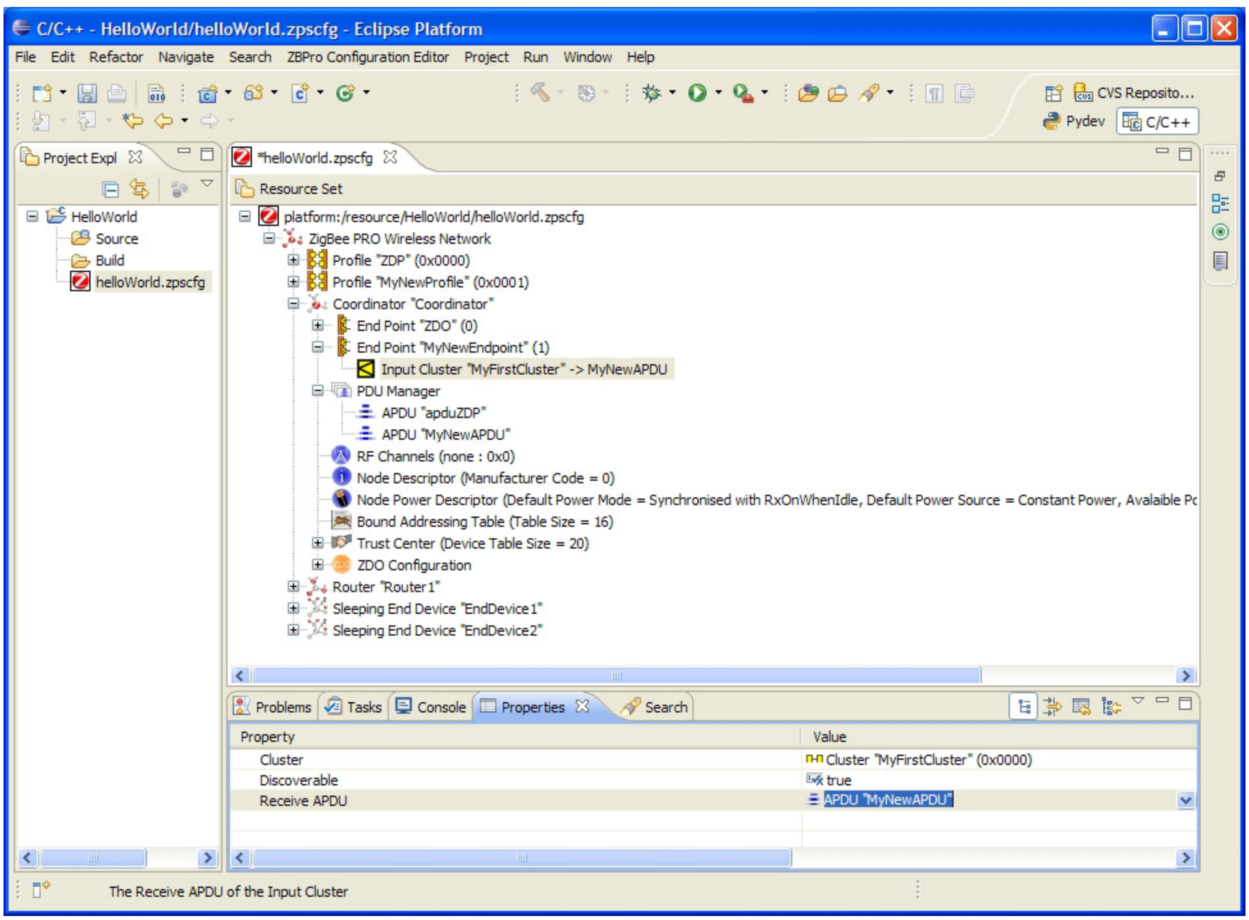


Figure 21: Input and Output Clusters

Step 5 Repeat Step 1 to Step 4 for Routers and End Devices, as required.

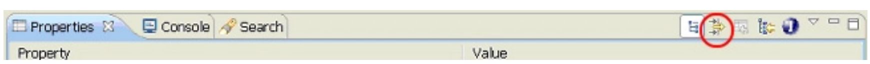
Note: In the above procedure, you may want to add the Default cluster (with a Cluster ID of 0xFFFF) as an input cluster. The inclusion of the Default cluster means that received messages that were intended for input clusters not supported by the endpoint will still be passed to the application. The messages must, however, come from defined application profiles, otherwise they are discarded.

12.4.4 Setting Advanced Device Parameters

You can set the advanced device parameters (detailed in Section 11.7) for a device as follows:

Step 1 Click on the relevant device (e.g. Coordinator) in the **Resource Set** pane.

Step 2 Click on the **Advanced Device Parameters** button in the tool bar of the lower pane (indicated below).



Step 3 Edit the relevant parameters in the **Properties** tab of the lower pane.

Step 4 Save your settings.

The ZigBee PRO R22 version of the stack allows the presence of multiple MAC interfaces. This is to support both 2.4G and 868 MHz frequency bands using the single ZigBee stack. To address this a MAC interface table needs to be configured in the ZPS Config diagram.

The MAC interface list can be found as an option for the node. For e.g., if you have ZigBee network with a router node. You can select the router node and press the right mouse button to provide the options. The MAC interface list can be found under **New Child > Mac Interface List**.

After adding the MAC interface list, select and right-click on the MAC interface list to provide the options. The MAC interface can be found under **New Child > MAC Interface**.

After adding the MAC interface, the properties can be updated. The default is 2.4G. This default can be kept. The “Router Allowed” properties should be set to “true”.

> **Note:** You will need to edit the advanced device parameters in order to change the Extended PAN ID (*APS Use Extended PAN ID* parameter) and the maximum number of children of the Co-ordinator or Router (*Active Neighbour Table Size* parameter) from the default values - see Section 5.1.1 and Section 5.1.2.

Part IV: Appendices

A. Handling Stack Events

The NXP ZigBee PRO stack events are listed below (they are detailed in Section 10.1):

ZPS_EVENT_NONE
ZPS_EVENT_APS_DATA_INDICATION
ZPS_EVENT_APS_DATA_CONFIRM
ZPS_EVENT_APS_DATA_ACK
ZPS_EVENT_NWK_STARTED
ZPS_EVENT_NWK_JOINED_AS_ROUTER
ZPS_EVENT_NWK_JOINED_AS_ENDDEVICE
ZPS_EVENT_NWK_FAILED_TO_START
ZPS_EVENT_NWK_FAILED_TO_JOIN
ZPS_EVENT_NWK_NEW_NODE_HAS_JOINED
ZPS_EVENT_NWK_DISCOVERY_COMPLETE
ZPS_EVENT_NWK_LEAVE_INDICATION
ZPS_EVENT_NWK_LEAVE_CONFIRM
ZPS_EVENT_NWK_STATUS_INDICATION
ZPS_EVENT_NWK_ROUTE_DISCOVERY_CONFIRM
ZPS_EVENT_NWK_POLL_CONFIRM
ZPS_EVENT_NWK_ED_SCAN
ZPS_EVENT_ZDO_BIND
ZPS_EVENT_ZDO_UNBIND
ZPS_EVENT_ZDO_LINK_KEY
ZPS_EVENT_BIND_REQUEST_SERVER
ZPS_EVENT_ERROR
ZPS_EVENT_APS_INTERPAN_DATA_INDICATION
ZPS_EVENT_APS_INTERPAN_DATA_CONFIRM
ZPS_EVENT_TC_STATUS

These events are handled by the following stack-supplied callback function:

void APP_vGenCallback(uint8 *u8Endpoint*, **ZPS_tsAfEvent** **psStackEvent***);**

The stack populates *psStackEvent* with a pointer to one of the above events, when it occurs.

Note 1: The above callback function must be incorporated in your application code, otherwise your application will not compile.

Note 2: You are recommended to push the stack events onto a message queue for delayed processing rather than process the event in the callback function.

B. Application Design Notes

This appendix collects together information and advice that will be useful to designers who are incorporating non-routine operations in their applications. The topics covered are:

- Fragmented data transfers - see Appendix B.1
- Sending data to sleeping End Devices - see Appendix B.2
- Clearing stack context data before a rejoin - see Appendix B.3
- Beacon filtering - see Appendix B.4
- Configuring ZigBee tables - see Appendix B.5
- Received message queues - see Appendix B.6
- Noise threshold for network formation - see Appendix B.7

B.1 Fragmented Data Transfers

The send 'with acknowledgement' functions (**ZPS_eAplAfUnicastAckDataReq()** and **ZPS_eAplAfUnicastIeeeAckDataReq()** and **ZPS_eAplAfBoundAckDataReq()**) allow a large data packet to be sent that may be fragmented into multiple messages/ frames during transmission. As a general rule, one of these two functions should be used when sending a data packet with a payload size greater than 80 bytes (note, however, that the use of APS security will reduce this limit, as payload bytes are taken up by security data). The processes of fragmentation at the sender and de-fragmentation at the receiver are transparent to the applications at the two ends, but the points described in the sub-sections below should be noted.

Note 1: Fragmentation is described further in Appendix B.2.2 in connection with fragmented data transfers to sleeping End Devices.

Note 2: The ZigBee network parameters referenced in this appendix are configured using the ZPS Configuration Editor and are described in Chapter 11. When setting up the APDUs to handle Rx fragmentation, care must be taken to ensure that the configuration setting in the ZPS Configuration Editor is sized to be able to handle 3* Tx Fragments.

B.1.1 Enabling/Disabling Fragmentation

In order to allow fragmented data transfers between two nodes, you must appropriately configure two ZigBee network parameters:

- Set the parameter *Maximum Number of Transmitted Simultaneous Fragmented Messages* to a non-zero value on the sending node, to allow transmitted messages to be fragmented.

- Set the parameter *Maximum Number of Received Simultaneous Fragmented Messages* to a non-zero value on the receiving node, to allow received fragmented messages to be re-assembled.

Note that setting either of these parameters to zero will disable the corresponding fragmentation feature but will reduce the size of your compiled application code.

B.1.2 Configuring Acknowledgements

You can configure how acknowledgements will be generated during a fragmented data transfer by setting the ZigBee network parameter *APS Max Window Size*, which must be set to the same value on the source and destination nodes. This parameter determines the number of fragments to be transferred before an acknowledgement is generated - for example, if a data packet is divided into 6 fragments and this parameter is set to 3, an acknowledgement will be generated after the third fragment and after the sixth fragment. Note that setting this parameter to a low value will result in a high level of network traffic, since a large number of acknowledgement packets are sent.

The acknowledgement for a group of fragments contains an indication of any missing fragments from the group, thus requesting the missing fragment(s) to be re-sent.

B.1.3 Acknowledgement Timeout

A timeout of approximately 1600 ms is applied to each acknowledgement, measured from the time at which the last data fragment in the relevant group was transmitted - if no acknowledgement is received within this timeout period, the entire group of fragments is automatically re-sent. Up to 3 more re-tries can subsequently be performed. For a fragmented data transfer, the time that elapses before a completely unacknowledged transmission is abandoned is difficult to estimate, since this time depends on the number of fragments, the network parameter *APS Max Window Size* and the network parameter *APS Inter-frame Delay* (time between transmissions of consecutive fragments).

B.2 Sending Data to Sleeping End Devices

As described in Section 5.5.3, data sent to a sleeping End Device is buffered in the node's parent until the End Device collects the data through a polling mechanism, typically on waking from sleep. It is important that the polling interval is not too long, as the buffered data will be discarded after 7 seconds. In addition, there is limited buffering space in the parent and the buffers are shared by all the children of the parent. Therefore, applications should be designed in such a way that data is only sent to a sleeping End Device when it is either awake or will wake in a timely manner to collect the data from its parent.

The following issues should also be considered when sending data to a sleeping End Device using one of the send 'with acknowledgement' functions: **ZPS_eAplAfUnicastAckDataReq()**, **ZPS_eAplAfUnicastIeeeAckDataReq()**, **ZPS_eAplAfBoundAckDataReq()**.

> **Note:** The ZigBee network parameters referenced in this appendix are configured using the ZPS Configuration Editor and are described in Chapter 11.

B.2.1 Acknowledged Data Transmission to Sleeping End Device

When data is sent and an acknowledgement is required from the receiver, a timeout of approximately 1600 ms is applied to the acknowledgement - if no acknowledgement is received by the sender within this timeout period, the data is automatically re-sent. Up to 3 more re-tries can subsequently be performed, totalling just over 3 seconds before the data transfer is finally abandoned.

In the case of data sent to a sleeping End Device, the acknowledgement is generated by the End Device after collecting the data from its parent. Thus, if the data is not collected within the acknowledgement timeout period, the data will be re-sent to the End Device (via its parent).

Note that if the buffered data is collected by the End Device after the final re-try by the sender but before the data is discarded by the parent (between approximately 3 and 7 seconds after the initial transmission), the acknowledgement that is eventually generated by the End Device will be ignored by the sender, since the transaction has already timed out and terminated.

B.2.2 Fragmented Data Transmission to Sleeping End Device

As explained in Section 5.5.1 and Appendix B.1, the send 'with acknowledgement' functions can be used to send large data packets that may need to be fragmented into multiple NPDUs during transmission. Therefore, when sending a fragmented data packet to a sleeping End Device, the issues described in Appendix B.2.1 apply.

In such a data transfer, the End Device should aim to collect all buffered data fragments from its parent before the transfer has completely timed out on the sender. Once the sender has abandoned the transaction, it will not respond to any acknowledgements requesting missing fragments (see Appendix B.1).

Once the End Device starts to receive fragmented data, it will stay awake until the transaction is complete and will run its own poll timer to automatically collect each fragment - the polling period for this timer is set through the ZigBee advanced device parameter *APS Poll Period*. This poll timer will run for the duration of the fragmented transaction and then stop. The responsibility for polling will then return to the application.

Sending fragmented data to a sleeping End Device is likely to result in duplicate fragments of the message being sent. A list of the last few fragments received, called the APS Duplicate table, is maintained in the End Device. This table allows new fragments to be compared with previous fragments and duplicates identified. The maximum number of entries (fragments) in this table can be configured through the network parameter *APS Duplicate Table Size*. This table size should not be made too small, as a short table will prevent duplicate fragments from being caught (4 may be a suitable value). This value should be considered in conjunction with the value of the network parameter *APS Persistence Time*, which represents the time for which resources associated with a message will be retained after the complete message has been received (once the resources have been released, they may be used for a new transaction) - during this period, any duplicate fragments that are received will be ignored.

B.3 Clearing Stack Context Data Before a Rejoin

If a node rejoins the same secured network (with ZigBee PRO security enabled) but its stack context data was cleared before the rejoin (by calling ***NvErase*()**), data sent by the node will be rejected by the destination node since the frame counter has been reset on the source node - frame counters are described in Section 1.8 and Appendix C. Sent data will be accepted again by the destination node when the frame counter for the source node reaches its last count known before the rejoin. Therefore, you are not recommended to clear the stack context data before a rejoin.

However, it is worth noting that frame counters are reset across the entire network when a new network key is broadcast by the Trust Centre using the function **ZPS_eAplZdoTransportNwkKey()** - see Section 5.8.4. Thus, if stack context data is cleared before a rejoin, the frame counter problem can be avoided by broadcasting a new network key from the Trust Centre (normally the Co-ordinator) immediately after the rejoin.

To restore the stack to a default state and not clear the frame counters **ZPS_vDefaultStack** should be used - see Section 6.1.1.

B.4 Beacon Filtering Guidelines

A filter can be introduced for filtering beacons in network searches (on a Router or End Device). Beacons can be filtered on the basis of Extended PAN ID (EPID), LQI value and device joining status/capacity (see below). The filter can be applied using the function **ZPS_bAppAddBeaconFilter()**.

If required, the above function must be called immediately before **ZPS_eAplZdoDiscoverNetworks()**, **ZPS_eAplZdoRejoinNetwork()** or **ZPS_eAplZdoStartStack()**.

A `tsBeaconFilterType` structure is supplied to the **ZPS_bAppAddBeaconFilter()** function in order to specify the details of the filter to be implemented, including:

- A blacklist or whitelist of networks in terms of a list of EPIDs
- The PAN ID of the network from which acceptable beacons should come
- The minimum LQI value of an acceptable beacon
- Flags indicating the properties on which beacons will be filtered, which include:
  - LQI value of beacon
  - Permit Join enabled on sending device
  - Capacity of sending device to accept Router children
  - Capacity of sending device to accept End Device children

After each discovery or rejoin, the flags will be cleared while all other fields of the structure will remain intact. The structure is detailed in Section 7.2.3.5.

The following general guidelines should be followed in using beacon filters:

- **Do not** implement a filter unless attempting a join, as this will prevent some stack operations from working correctly
- **Do not** enable a blacklist and whitelist at the same time
- **Do not** declare your filter structure as a local variable in a function, as it needs to exist for the duration of the discovery

The following guidelines are relevant to network rejoins and associations.

Network Rejoin

- **Do** set up a whitelist containing a single EPID corresponding to the network that the node is to rejoin (if only one network is of interest) and/or the PAN ID of this network
- **Do** set up an LQI filter to reject distant beacons, if required
- **Do not** enable filtering on Permit Join or Router/End Device Capacity

Association

- **Do** set up an LQI filter to reject distant beacons, if required
- **Do** filter on the Permit Join status to only find potential parents and networks that are accepting association requests
- **Do** filter on Router/End Device Capacity, if required, depending on device type

> **Note:** A blacklist can be built up over several attempts to discover and associate, by keep adding to the array of EPIDs as each network is rejected.

B.5 Table Configuration Guidelines

This section provides guidelines on configuring various tables used by the ZigBee PRO stack. These tables can be configured through ZigBee network parameters in the ZPS Configuration Editor. The tables are sized, by default, to support a network of up to 250 nodes. The table sizes can be increased to support more nodes, but this will be at the expense of RAM and/or Flash usage.

The tables and their configuration are individually described in the sections below, which reference to the ZigBee network parameters used to configure the table sizes (the network parameters are detailed in Chapter 11).

B.5.1 Neighbour Table

The Neighbour table on a routing node (Router or Co-ordinator) holds information about the node's immediate neighbours:

- The first entry in the table contains information about the node's parent
- Part of the table holds information about child nodes which have joined the network through the local device
- The rest of the table holds information about nodes which are neither children nor the parent (these 'other' nodes are only relevant to Mesh networks)

The Neighbour table size is, by default, set to 26 - this is the minimum size required for a ZigBee-Compliant Platform. The table size may be increased through the parameter *Active Neighbour Table Size* to reflect the density of the network, but increasing the table size will use more RAM. Increasing the Neighbour table size beyond 26 will also result in an extra link status packet (since one of these packets can accommodate a maximum of 26 neighbours), thus doubling the traffic for these periodic packets.

The first two parts of the Neighbour table, for the device's parent and children, form a sub-table that is persisted in Flash. This sub-table must not occupy more than two-thirds of the Neighbour table. Since this sub-table contains child entries, the size of the sub-table determines the number of children that the device is allowed to have - the maximum number of children is one less than the sub-table size. The default size for the sub-table is 5, allowing up to 4 child nodes, but the size can be changed through the parameter *Child Table Size* (which corresponds to the total number of sub-table entries including the parent's entry, not just the child entries). Note that increasing the sub-table size will use more Flash for persisted data.

B.5.2 Address Map Table

The Address Map table on a node is used to keep a record of the address-pairs of network nodes with which the local node needs to communicate directly - that is, the IEEE/MAC address and network address of each of these nodes. In fact, an Address Map table entry only contains an index to an entry in the MAC Address table, where the actual addresses of the node are stored (see Appendix B.5.3). The population of these tables is done as the result of device announcement messages.

The default size of the Address Map table is 10, but the size can be changed through the parameter *Address Map Table Size*. The Address Map table is fully persisted in Flash. Therefore, increasing the size of this table will impact both RAM and Flash usage.

B.5.3 MAC Address Table

The MAC Address table on a node is used to store the address-pairs of other network nodes - that is, the IEEE/MAC address and network address of each of these nodes. The entries in the MAC Address table are referenced from entries of both the Neighbour table and Address Map table. Therefore, the MAC Address table should be sized according to the combined sizes of the Neighbour table and Address Map table.

The default size of the MAC Address table is 36, but the size can be changed through the parameter *Maximum Number of Nodes*. The MAC Address table is fully persisted in Flash. Therefore, increasing the size of this table will impact both RAM and Flash usage.

B.5.4 Routing Table

A Routing table is held by the Co-ordinator and Router nodes to store routing information to other nodes in the network.

The default size of the Routing table is 70, which should be sufficient for most applications, but the size can be changed through the parameter *Routing Table Size*. The table size should be increased if routing bottlenecks are observed. The Co-ordinator needs to store routes to all the nodes in the network if it is required to communicate with every node - in this case, the Routing table size should be increased to the size of the network.

The Routing table is not persisted and any increase will therefore only affect RAM usage.

B.5.5 Broadcast Transaction Table

The Broadcast Transaction table is used for the origination, processing and passive acknowledgement of broadcast transmissions. The minimum required size of this table for a ZigBee-Compliant Platform is 9. However, an application that produces a large number of broadcasts may need a larger table. The size of the table can be set through the parameter *Broadcast Transaction Table Size*.

B.5.6 Route Discovery Table

The Route Discovery table is used to hold temporary details of a route discovery transaction. The table size dictates how many individual route discoveries can occur on the local node at a given time. The default size of the Route Discovery table is 2, but the size can be changed through the parameter *Route Discovery Table Size*. The default value severely restricts the number route discoveries and hence broadcasts on the network. Increasing the table size also requires increases in the Routing table and Broadcast Transaction table sizes.

The Route Discovery table is not persisted and any increase will therefore only affect RAM usage.

B.5.7 Discovery Table

A Discovery table is held by the Router and End Device nodes to store the results of a channel scan when searching for a network to join. The default size of the Discovery table is 8, but the size can be changed through the parameter *Discovery Neighbour Table Size*.

B.5.8 Route Record Table

The Route Record table is only relevant to a device which will be the concentrator in a network, if many-to-one routing is implemented. This table replaces the Routing table in the node.

The size of the Route Record table can be set through the parameter *Route Record Table Size*. In the concentrator node, this table size should be set to the size of the network. Since this table then replaces the Routing table in the node, the Routing table size should be set to 1 (see Appendix B.5.4). In all other network nodes, the size of the Route Record table should be set to 1.

B.6 Received Message Queues

All messages received on a network node are pushed into one of the following two queues:

- ZPS_msgMcpsDcfm
- ZPS_msgMcpsInd

These queues must be created by the application using the function **ZQ_vZQueueCreate()**, as indicated in the example code in Section 5.9.1.2.

ZPS_msgMcpsDcfm

All IEEE 802.15.4 MAC data deferred confirm events are added to this queue. The default size of this queue is 8 but a different queue size can be set when the queue is created. The queue can overflow if there is heavy network traffic.

ZPS_msgMcpsInd

All IEEE 802.15.4 MAC data packets are added to this queue. The default size of this queue is 24 but a different queue size can be set when the queue is created. The queue can overflow if there is heavy network traffic.

B.7 Noise Threshold for Forming a Network

The ZigBee PRO stack provides a mechanism for forming a new network in the quietest IEEE802.15.4 radio channel. The Co-ordinator (centralised network) or Router (distributed network) that forms the network performs a channel scan to listen for activity from other local networks.

During the channel scan, the activity in each channel is assigned a noise level in the range 0 to 254. This result is compared with a noise level threshold, which is defined by the NIB value `u8VsFormEdThreshold` (which is part of the structure `ZPS_tsNwkNibInitialValues`). If the measured noise level for a channel is above this threshold, the channel will be excluded from further consideration. Therefore, if all the channels in the scan are noisier than the threshold allows, no network will be formed.

The stack then re-scans the channels that passed the noise threshold test (if any) and selects the one with the lowest beacon count in which to form the network. Note that:

- This assessment takes into account IEEE802.15.4 beacons only and no activity from networks based on other systems, such as Wi-Fi
- The assessment is based on beacons only and does not consider the noise levels of the shortlisted channels

Default Behaviour

To avoid the situation in which no network is formed, the default value of `u8VsFormEdThreshold` is 0xFF, which is a special value and not a noise threshold. In this case, the network is always formed in the channel with the lowest IEEE802.15.4 beacon activity (no noise level assessment is performed).

Customising the Scan

You can implement network formation based on the noise level threshold, as described above, by setting `u8VsFormEdThreshold` to an appropriate value in the range 0 to 254. In the following code fragment, a noise level threshold of 100 is set:

```
ZPS_psNwkNibGetHandle(ZPS_pvAplZdoGetNwkHandle())->u8VsFormEdThreshold = 100
```

Thus, in the above case, all channels with a noise level above 100 will be rejected.

If no suitable channel is found and no network formed, the application can dynamically increase the value of `u8VsFormEdThreshold` and initiate another scan.

C. Implementation of Frame Counters

Frame counters are used to prevent message replay attacks (see Section 1.8). Each message sent across the network carries a frame count. It is inserted by the sending node and is part of a sequence of frame counter values for messages from this node.

Two types of frame counter exist:

- **NWK frame counters:** These are implemented automatically at the NWK layer of the stack and are used in all network communications.
- **APS frame counters:** These are optional and applied by the APS layer of the stack to one or more specific source/destination links.

The rest of this section describes the standard NWK frame counters. Detailed information on NWK and APS frame counters can be found in the ZigBee 3.0 specification.

A node locally maintains two types of NWK frame counter:

- **Outgoing frame counter:** A single frame counter for all sent messages is maintained in RAM. It is incremented every time a message is transmitted (to any destination node) and its value is inserted in the message. Its value can be persisted in non-volatile memory - the NXP method of persisting this counter is described later.
- **Incoming frame counters:** A frame counter is maintained in RAM for every node in the local Neighbour table. It records the frame count contained in the last message received from this node. Its value is not persisted, as persisting the frame counters for all neighbours would use too much space in non-volatile memory. On a power-cycle or reset, these values are all reset to 0.

If a destination node receives a message containing a frame count which is less than the locally held incoming frame counter value for the same source node (corresponding to the frame count of the previous message received from the source node), it rejects the message. The receiving node informs the rest of the network by sending out a Network Status command with a status value of 0x11 (bad frame counter).

A problem may occur if the outgoing frame counter is not persisted and reverts to 0 when the node is reset. This will result in messages being sent with a frame count lower than the value expected by the destination node(s). Consequently, the messages from this node will be rejected.

Persisting the outgoing frame counter is therefore advisable, but doing this on every update of the counter causes excessive Flash wear. To minimise this wear, the NXP stack software only persists the outgoing frame counter every time it reaches a multiple of 1024. If the node is reset, the current frame counter value is likely to be greater than the persisted value, so 1024 is added to the recovered value. This ensures that the outgoing frame counter will always be larger than the value in the last message sent before the reset. For example, if the last message transmitted had a frame count of 4000 and then the node is reset, the last persisted value of 3072 will be recovered but 1024 will be added to it to make an outgoing frame counter of 4096 (which will be persisted).

D. Storing Applications in Device Flash Memory

During start-up, the device bootloader (provided in internal ROM) searches for a valid application image in internal Flash memory. If one is present then the device will boot directly from Flash memory. If no image is found then the bootloader will drop into in system programming (ISP) mode.

E. Glossary

Term	Description
Address	A numeric value that is used to identify a network node. In ZigBee, the device’s 64-bit IEEE/MAC address or 16-bit network address is used.
AIB	APS Information Base: A database for the Application Support (APS) layer of the ZigBee stack, containing attributes concerned with system security.
APDU	Application Protocol Data Unit: Part of a wireless network message that is handled by the application and contains user data.
API	Application Programming Interface: A set of programming functions that can be incorporated in application code to provide an easy-to-use interface to underlying functionality and resources.
APS	Application Support: A sub-layer of the Application layer of the ZigBee stack, relating to communications with applications, binding and security.
Application	The program that deals with the input/output/processing requirements of the node, as well as high-level interfacing to the network.
Application Profile	A collection of device descriptors that characterise an application for a particular market sector. An application profile can be public or private. A public profile is identified by a 16-bit number, assigned by the ZigBee Alliance.
Attribute	A data entity used by an application, e.g. a temperature measurement. It is part of a ‘cluster’ along with a set of commands which can be used to pass attribute values between applications or modify attributes.
Binding	The process of associating an endpoint on one node with an endpoint on another node, so that communications from the source endpoint are automatically routed to the destination endpoint without specifying addresses.
Channel	A narrow frequency range within the designated radio band - for example, the IEEE 802.15.4 2400-MHz band is divided into 16 channels. A wireless network operates in a single channel which is determined at network initialisation.
Child	A node which is connected directly to a parent node and for which the parent node provides routing functionality. A child can be an End Device or Router. Also see Parent.
Cluster	A collection of attributes and commands associated with the endpoint for an application. The commands are used to communicate or modify attribute values. A cluster has input and output sides - an output cluster issues a command which is received and acted on by an input cluster.
Context Data	Data which reflects the current state of the node. The context data must be preserved during sleep (of an End Device).
Co-ordinator	The node through which a network is started, initialised and formed - the Co-ordinator acts as the seed from which the network grows, as it is joined by other nodes. The Co-ordinator also usually provides a routing function. All networks must have one and only one Co-ordinator.
End Device	A node which has no networking role (such as routing) and is only concerned with data input/output/processing. As such, an End Device cannot be a parent but can sleep to conserve power.

Term	Description
Endpoint	A software entity that acts as a communications port for an application on a ZigBee node. A node can support up to 240 endpoints, numbered 1 to 240. Two special endpoints are also supported - endpoint 0 is used by the ZDO and endpoint 255 is used for a broadcast to all endpoints on the node.
Extended PAN ID (EPID)	A 64-bit identifier for a ZigBee PRO network that is assigned when the network is started. A value can be pre-set or, alternatively, the IEEE/MAC address of the Co-ordinator can be used as the EPID.
IEEE 802.15.4	A standard network protocol that is used as the lowest level of the ZigBee software stack. Among other functionality, it provides the physical interface to the network's transmission medium (radio).
IEEE/MAC Address	A unique 64-bit address that is allocated to a device at the time of manufacture and is retained by the device for its lifetime. No two devices in the world can have the same IEEE/MAC address.
Joining	The process by which a device becomes a node of a network. The device transmits a joining request. If this is received and accepted by a parent node (Co-ordinator or Router), the device becomes a child of the parent. Note that the parent must have "permit joining" enabled.
Mesh Network	A wireless network topology in which all routing nodes (Routers and the Co-ordinator) can communicate directly with each other, provided that they are within radio range. This allows optimal and flexible routing, with alternative routes if the most direct route is not available.
Network Address	A 16-bit address that is allocated to a ZigBee node when it joins a network. The Co-ordinator always has the network address 0x0000. In IEEE 802.15.4 terminology, it is called the short address.
NIB	NWK Information Base: A database containing attributes needed in the management of the Network (NWK) layer of the ZigBee stack.
Node Descriptor	A set of information about the capabilities of a node.
Node Power Descriptor	A set of information about a node's current and potential power supply.
NPDU	Network Protocol Data Unit: The transmitted form of a wireless network message (incorporates APDU and header/footer information from stack).
PAN ID	Personal Area Network Identifier: This is a 16-bit value that uniquely identifies the network - all neighbouring networks must have different PAN IDs.
Parent	A node which allows other nodes (children) to join the network through it and provides a routing function for these child nodes. A parent can be a Router or the Co-ordinator. Also see Child.
Router	A node which provides routing functionality (in addition to input/output/processing) if used as a parent node. Also see Routing.
Routing	The ability of a node to pass messages from one node to another, acting as a stepping stone from the source node to the target node. Routing functionality is provided by Routers and the Co-ordinator. Routing is handled by the network level software and is transparent to the application on the node.
Simple Descriptor	A set of assorted information about a particular application/endpoint.

Term	Description
Sleep Mode	An operating state of a node in which the device consumes minimal power. During sleep, the only activity of the node may be to time the sleep duration to determine when to wake up and resume normal operation. Only End Devices can sleep.
Stack	The hierarchical set of software layers used to operate a system. The high-level user application is at the top of the stack and the low-level interface to the transmission medium is at the bottom of the stack.
Stack Profile	The set of features implemented from the ZigBee specification - that is, all the mandatory features together with a subset of the optional features. The ZigBee Alliance define two Stack Profiles for use with public Application Profiles - ZigBee and ZigBee PRO.
UART	Universal Asynchronous Receiver Transmitter: A standard interface used for cabled serial communications between two devices (each device must have a UART).
User Descriptor	A user-defined description of a node (e.g. “KitchenLight“).
ZigBee Base Device	A framework for the use of ZigBee device types that provides basic functionality such as commissioning. Its functionality is defined in the ZigBee Base Device Behaviour (BDB) specification from the ZigBee Alliance.
ZigBee Certified Product	An end-product that uses ZigBee Compliant Platforms and public Application Profiles, and which has been tested for ZigBee compliance and subsequently authorised to carry the ZigBee Alliance logo.
ZigBee Cluster LIbrary (ZCL)	A collection of clusters that can be individually employed in ZigBee devices, as required, to implement the functionality of a device.
ZigBee Compliant Platform	A component (such as a module) that has been tested for ZigBee compliance and authorised to be used as a building block for a ZigBee end-product.
ZigBee Device Objects (ZDO)	A special application which resides in the Application Layer on all nodes and performs various standard tasks (e.g. device discovery, binding). The ZDO communicates via endpoint 0.

Revision History

Version	Date	Comments
1.0	19 June 2018	First release
2.0	18 November 2019	Updated for K32W\JN5189

Important Notice

NXP Semiconductors

For online support resources and contact details of your local NXP office or distributor, refer to:

www.nxp.com