

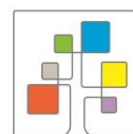


JenNet-IP Application Template Application Note

JN-AN-1190

Public v1059

04/09/2013



JenNet-IP

Contents

About this Manual	5
Organisation	5
Conventions	6
Acronyms and Abbreviations	6
Compatibility	6
Related Documents	7
Trademarks	7
Certification	7
1 Introduction	8
2 Application Concepts	9
2.1 Gateway System Topology	9
2.1.1 Gateway Hardware	11
2.2 Standalone System Topology	12
2.2.1 Gateway to Standalone Mode Failover	13
2.3 MIBs and Variables	13
2.4 Identifiers	14
2.4.1 Device ID (32 bits)	14
2.4.2 Device Type IDs (16 bits)	15
2.4.3 MIB IDs (32 bits)	16
2.5 Message Transmission	17
2.5.1 Unicast Messaging	17
2.5.2 Multicast Messaging	17
3 System Operation	18
3.1 Gateway System Operation	19
3.1.1 Gateway System Operation Overview	20
3.1.2 Setting Up the Gateway System	23
3.1.3 Operating the Gateway System	31
3.1.4 Group Configuration and Control	39
4 MIB Variable Reference	43
4.1 Node MIBs	44
4.1.1 NodeStatus MIB (0xFFFFFE80)	44
4.1.2 NodeConfig MIB (0xFFFFFE81)	49
4.1.3 NodeControl MIB (0xFFFFFE82)	50
4.2 Network MIBs	52
4.2.1 NwkConfig MIB (0xFFFFFE89)	52
4.2.2 NwkProfile MIB (0xFFFFFE8D)	57
4.2.3 NwkStatus MIB (0xFFFFFE88)	61
4.2.4 NwkControl MIB (0xFFFFFE8A)	64
4.2.5 NwkSecurity MIB (0xFFFFFE8B)	65
4.2.6 NwkTest MIB (0xFFFFFE8C)	69
4.3 Peripheral MIBs	76
4.3.1 AdcStatus MIB (0xFFFFFE90)	76
4.3.2 DioStatus MIB (0xFFFFFE70)	79
4.3.3 DioConfig MIB (0xFFFFFE71)	81
4.3.4 DioControl MIB (0xFFFFFE72)	93
5 Software Reference – PROOF READ TO HERE	96
5.1 DeviceTemplate	97
5.1.1 Makefile	98
5.1.2 DeviceDefs.h	103

5.1.3 DeviceTemplate.c	104
5.2 DeviceDio	110
5.2.1 Makefile	110
5.2.2 DeviceDefs.h	111
5.2.3 DeviceDio.c	112
5.3 Common Software	114
5.3.1 Config.h	114
5.3.2 Node.h, Node.c	114
5.3.3 AHI_EEPROM.h, AHI_EEPROM.c	122
5.3.4 Exception.h, Exception.c	122
5.3.5 Security.h, Security.c	122
5.3.6 Address.h, Address.c	123
5.3.7 Uart.h, Uart.c	123
5.4 MibCommon Library	124
5.4.1 Changing the Library	124
5.4.2 Organisation	125
5.4.3 Node MIB	127
5.4.4 Groups MIB	128
5.4.5 NodeStatus MIB	129
5.4.6 NodeControl MIB	130
5.4.7 NwkConfig MIB	131
5.4.8 NwkProfile MIB	135
5.4.9 NwkStatus MIB	137
5.4.10 NwkSecurity MIB	139
5.4.11 NwkTest MIB	144
5.4.12 AdcStatus MIB	146
5.5 MibDio Modules	149
5.5.1 Changing the Modules	149
5.5.2 Organisation	149
5.5.3 DioConfig MIB	151
5.5.4 DioStatus MIB	153
5.5.5 DioControl MIB	154
Appendices	156
A Revision History	156

About this Manual

This manual provides information about the **JenNet-IP Application Template Application Note**. This Application Note provides template source code for creating Smart Devices that operate in a low power Wireless Personal Area Network (WPAN). These Smart Devices can be monitored and controlled using the standard Internet Protocol (IP) from within the WPAN, externally from a Local Area Network (LAN) and also from a Wide Area Network (WAN) such as the internet.

The design of the source code is covered in detail to provide enough information for developers to add to the code in order to develop different Smart Devices. Developers writing applications for devices within the WPAN will find this information useful.

The Management Information Bases (MIBs) and variables implemented in the devices in this Application Note are covered. These allow the devices within the WPAN to be monitored and controlled. Developers writing applications to control devices within the WPAN from inside or outside the WPAN will find this information useful.

Organisation

This manual consists of the following chapters:

- [Introduction](#) provides an overview of the Application Note
- [Application Concepts](#) describes the features of the Application Note at a high level.
- [System Operation](#) describes how to operate the devices in the Application Note as an end user.
- [MIB Variable Reference](#) describes in detail the MIBs and variables implemented in the devices in this Application Note. These allow devices within the WPAN to be monitored and controlled. Developers writing advanced applications to monitor and control devices in the WPAN from devices inside or outside the WPAN should refer to this chapter.
- [Software Reference](#) describes the source code in detail. Developers that want to adapt the existing devices or create new devices that operate within the WPAN should refer to this chapter.

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 typeface.

Acronyms and Abbreviations

The following acronyms and abbreviations are used in this document:

API	Application Programming Interface
DIO	Digital Input/Output
IP	Internet Protocol
LAN	Local Area Network
LED	Light Emitting Diode
MIB	Management Information Base
OND	Over Network Download
PDM	Persistent Data Manager
SDK	Software Development Kit
WAN	Wide Area Network
WPAN	Wireless Personal Area Network

Compatibility

The software provided with this Application Note has been tested with the following Evaluation Kits and SDK versions. The SDK installers are available from the NXP Wireless Connectivity Techzone JenNet-IP webpage:

<http://www.nxp.com/techzones/wireless-connectivity/jennet-ip.html>

Product Type	Part Number	Public Version	Internal Version	Supported Chips
SDK Toolchain	JN-SW-4041	v1.1	-	JN5168 JN5164
JN516x Evaluation Kit	JN516x EK001	-	-	JN5168 JN5164
JenNet-IP JN516x SDK Libraries	JN-SW-4065	-	v940	JN5168 JN5164

Related Documents

The following documents provide further information on the hardware and software used in this Application Note. They can be downloaded from the NXP Wireless Connectivity TechZone JenNet-IP webpage:

<http://www.nxp.com/techzones/wireless-connectivity/jennet-ip.html>

JN-UG-3093 JN516x EK001 Evaluation Kit User Guide

Provides information on how to operate the JN516x Evaluation Kit.

JN-UG-3007 JN51xx Flash Programmer User Guide

Provides instructions on connecting Evaluation Kit boards and programming them with embedded firmware.

JN-UG-3080 JenNet-IP WPAN Stack User Guide

Provides detailed information on the concepts and operation of the JenNet-IP WPAN network stack. This includes reference information for the functions, structures and variables that create the JenNet-IP WPAN Stack APIs that were used to create the applications in this Application Note.

JN-UG-3086 JenNet-IP LAN/WAN Stack User Guide

Provides detailed information on creating applications to access JenNet-IP devices via a LAN or WAN.

JN-UG-3087 JN516x Integrated Peripherals API

Provides information on the APIs used to program the JN516x on-chip peripherals.

JN-UG-3064 SDK Installation and User Guide

Provides information on installing and using the Software Development Kits.

JN-AN-1110 JenNet-IP Border-Router Application Note

Provides source code for the JenNet-IP Border-Router.

JN-AN-1162 JenNet-IP Smart Home Application Note

Provides additional JenNet-IP device examples based upon this template. These examples are focused upon a Smart Lighting system.

Trademarks

“JenNet”, “JenNet-IP” and the tree icon are trademarks of NXP B.V..

Certification

In order to use the JenNet-IP trademark and logo on a JenNet-IP product, the product must be certified. This is to ensure that the product correctly supports the JenNet-IP protocol and that JenNet-IP products will interoperate with each other. It is possible to use the JenNet-IP software stack on non-certified products but, in this case, the JenNet-IP trademark and logo cannot be displayed on the product. For further information, see **www.JenNet-IP.com**.

1 Introduction

This Application Note provides template software for developing Smart Devices to operate in a network. This allows the control and monitoring of Smart Devices via standard Internet Protocol messages over a low power radio network.

Smart Devices can be controlled and monitored from within the low power radio network and also, with the addition of a JenNet-IP Gateway, from a standard Local Area Network and even a Wide Area Network such as the internet.

The Application Note includes software for the following Smart Devices:

- Template software providing the minimum functionality required for devices to join and maintain their place in a low power wireless network. Developers can add additional software to the template to create a wide variety of Smart Devices.
- Digital Input/Output (DIO) software allowing the DIO pins of the JN516x devices to be configured monitored and controlled in a generic manner. This can be used for simple prototyping of devices using Digital Input and Output. It also serves as an example of how create new Smart Devices from the template.

2 Application Concepts

JenNet-IP networks can operate in one of two modes:

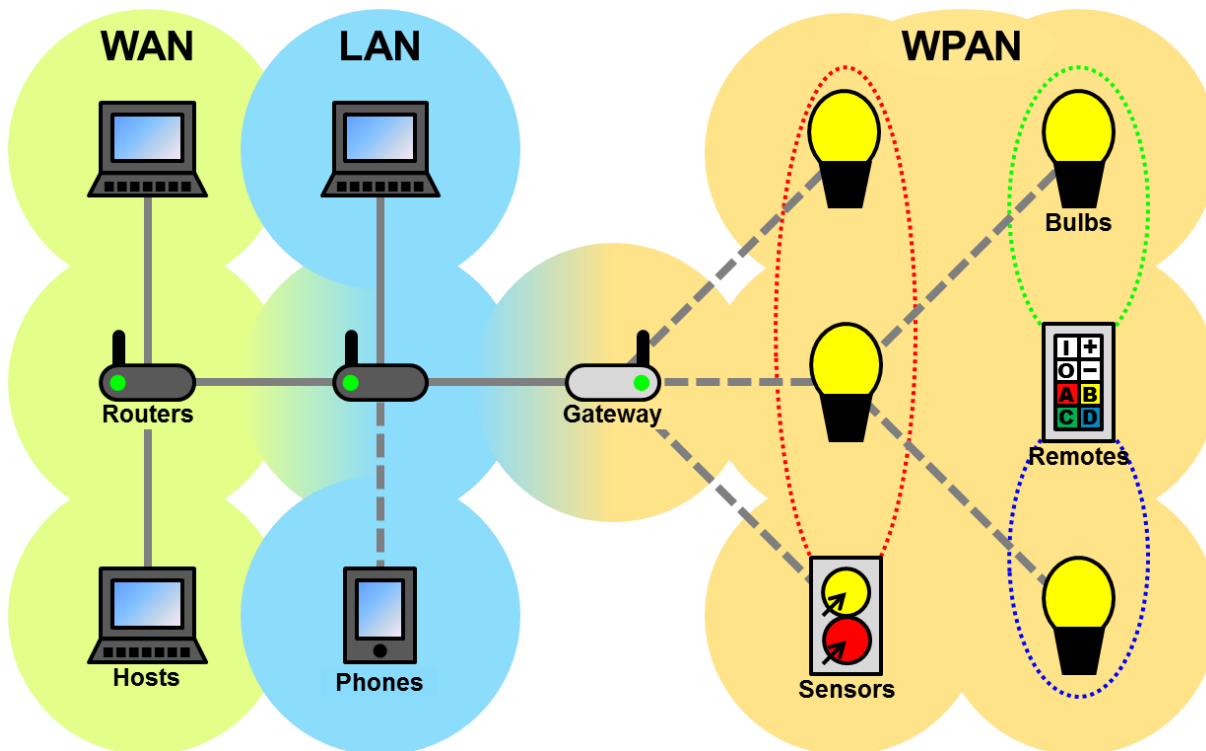
- Gateway Mode includes a gateway device allowing access to the low-power wireless Smart Devices from other Internet Protocol devices connected via the Local IP Network, Wi-Fi or even from the external Internet. Smart Devices can also be controlled by other Smart Devices within the low power wireless network such as Remote Controls. This system provides the most flexibility and options for controlling and monitoring Smart Devices.
- Standalone Mode does not include a Gateway Device. The Smart Devices form a low power wireless network that can only be controlled by other Smart Devices from within the network such as the Remote Control included in the JN-AN-1162 JenNet-IP Smart Home Application Note. This type of system provides a low cost entry point for building a Smart Device system while allowing a Gateway Device to be added later.



Note the examples in this section are taken from the **JN-AN-1162 Smart Home Application Note** as they provide good example of a complete JenNet-IP system.

2.1 Gateway System Topology

The following diagram shows the topology of a typical gateway system built from the lighting devices in the **JN-AN-1162 Smart Home Application Note**:



The components of the Smart Home lighting system are as follows:

The Router, provides Internet Protocol routing services for devices in the network. This provides standard IP routing of packets in the LAN and WAN domains via a standard internet router device.

Adding a JenNet-IP Gateway device to the internet router extends the IP network into the WPAN domain providing low power wireless access to the Smart Device network. The JenNet-IP Gateway includes a Border Router device, (either internally or externally), which provides the WPAN radio services.

Bulbs, allow wireless control of lighting in the home. These devices act as router nodes in the low power JenNet-IP wireless network extending the network for other Smart Devices to join.

Remote Controls, allow control of other Smart Devices in the low power JenNet-IP wireless network. These devices operate as sleeping broadcaster devices in order to allow mobile operation and preserve battery life. To do this they spend most of their time asleep preserving power and only waking to read button inputs. Commands are always broadcast to a group of devices so the remote does not need to maintain a full connection to the network, this allows the remote to be freely moved around the area covered by the WPAN.

Sensors, monitor occupancy and light levels in an area and can control the bulbs based upon their readings.

In a gateway system the Smart Devices form a JenNet-IP tree network allowing messages to be directed to both individual nodes in the form of unicasts and groups of nodes in the form of broadcasts.

2.1.1 Gateway Hardware

The Internet Router / JenNet-IP Gateway hardware can be formed by a number of different hardware and software combinations:

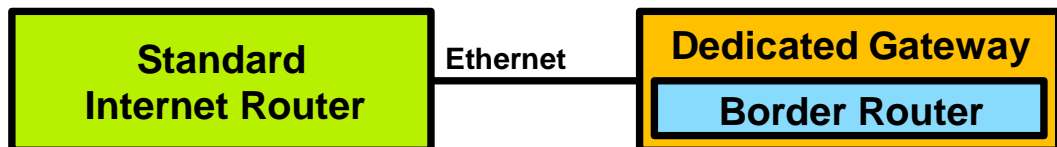
2.1.1.1 Dedicated Gateway

The Gateway device normally includes a CPU that controls the internal JN5168 low power radio Border Router device via a serial interface.

The Gateway CPU is responsible for providing authentication and Over-Network Download services. It may also provide GUIs in the form of webpages and/or provide translation of IPv4 to IPv6 packets.

The internal Border Router is responsible for providing JenNet-IP WPAN radio services to the Gateway CPU controlled by the JenNet-IP serial protocol.

The advantage of this configuration is that existing IP based systems can be simply extended to include JenNet-IP devices by simply connecting the Dedicated Gateway to an existing LAN Router with an Ethernet cable.



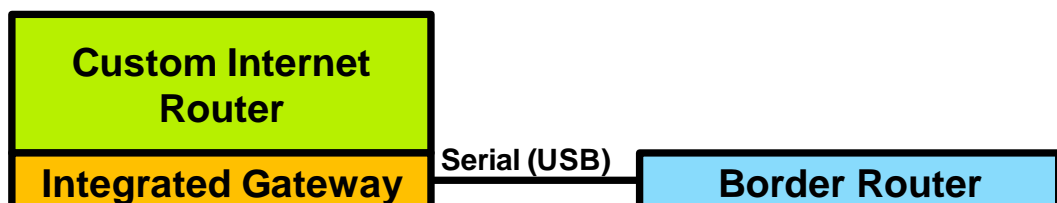
2.1.1.2 Custom Internet Router / USB Border Router Dongle

This configuration replaces the firmware in a standard internet router with custom firmware based upon OpenWRT. The processor in the Internet Router is used to provide the authentication and Over-Network Download services in addition to providing the standard Internet Routing services. The custom firmware may also use the processor to provide GUIs in the form of webpages or provide translation of IPv4 to IPv6 packets.

The JN5168 low power radio device is added to the system in the form of a simple dongle connected via USB to the Internet Router. This provides a serial connection that can be used by the custom firmware to communicate with the JenNet-IP low power wireless network.

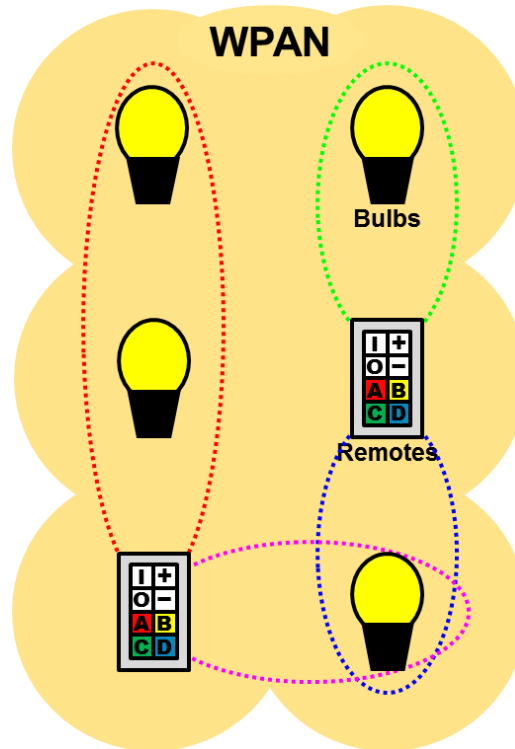
The advantage of this configuration is that the majority of both the Internet Routing and Gateway functionality is encapsulated within a single device, however the need to program custom firmware results in a less user-friendly experience.

This configuration is provided in the **JN516x EK001 Evaluation Kit**.



2.2 Standalone System Topology

The following diagram shows the topology of a standalone system built from the lighting devices in JN-AN-1162 Smart Home Application Note:



This topology does not include a Coordinator node to form the network. Instead a Remote Control chooses a security key for the network and can be placed into a commissioning mode using a sequence of keys. While in commissioning mode other Smart Devices in range can communicate with the Remote Control to retrieve the security key, other network settings and join the network.

Once Smart Devices are members they may be controlled by Remote Controls and other devices in the system.

When in standalone mode the Smart Devices do not form a tree network but instead only accept broadcast commands and re-broadcast them for other standalone Smart Devices to receive, only devices that are in the broadcast group the command is addressed to will act upon the command.

2.2.1 Gateway to Standalone Mode Failover

When devices in a gateway network lose contact with the gateway network they will continue to receive broadcast commands and so operate as standalone devices. While in this mode they attempt to re-join the network (possibly with a different parent).

This allows Smart Devices to be controlled from other devices running within the WPAN, (such as remote controls and sensors), even while not in the tree network. This situation may occur if a node or border router device is powered off.

Devices that are power cycled when they are in a gateway network will start up in standalone mode allowing them to accept broadcast commands. The border router regularly broadcasts messages to the network to indicate its presence. Devices that have powered up into standalone mode will attempt to re-join the tree network upon receiving one of these messages. This provides self-healing of the network if nodes are turned off.

2.3 MIBs and Variables

The functionality of the Smart Devices is implemented by a set of Management Information Bases (MIBs). Each MIB provides a set of variables that allow the device to be monitored and controlled. Each MIB groups together a set of variables that provides access to a particular function of the device.

Where different devices implement the same functionality they do so via the same set of MIBs. Therefore some MIBs are common to many devices while other MIBs are specific to certain device types.

The software included in this Application Note may be re-used to program additional devices types. The included code and common MIBs may be used unchanged in order to introduce new devices to a network. The tasks for a developing a new device type will then typically include some combination of the following:

- For a device to be remotely monitored – appropriate MIBs and variables need to be created with the variables being set to appropriate values for the data being monitored.
- For a device to remotely monitor other devices – it must identify the devices to be monitored in the network then read the variables to obtain the data being monitored. Similar functionality is required when writing applications to monitor devices from outside the WPAN.
- For a device to be remotely controlled – appropriate MIBs and variables need to be created. When the variables are written to by remote devices appropriate actions should be taken to respond to the command.
- For a device to remotely control other devices – it must identify the devices to be controlled in the network then write to the appropriate variables to control the device. Similar functionality is required when writing applications to control devices from outside the WPAN.

The [MIB Variable Reference](#) section of this Application Note covers each MIB and variable in detail for comprehensive information.

2.4 Identifiers

Various identifiers are used to identify devices, manufacturers, products and MIBs. These are described in the following sections:

2.4.1 Device ID (32 bits)

The 32-bit Device ID is used to identify different devices. Devices with the same Device ID *must* contain identical MIBs.

- ! *The Device ID is often used to cache information about the MIBs that are present in a device, therefore if the MIBs are changed a new Device ID should be allocated.*

! *When changing the MIBs in a device during development it may be necessary to power down changed devices and reset the gateway to clear any information cached in the gateway for the changed device.*

The Device IDs are made available in the DeviceID MIB present in each device.

The Device ID is divided into three components:

2.4.1.1 Sleeping Device Flag (1 bit)

The most significant bit is used to identify if a device is a sleeping End Device. Setting the bit indicates a sleeping End Device.

Software communicating with an End Device may request an End Device to stay awake receive further messages, this bit can be used to identify such devices.

2.4.1.2 Manufacturer ID (15-bits)

The next most significant 15 bits are the Manufacturer ID which identifies the manufacturer of a device. All the devices in the Application Note use NXP's Manufacturer ID of 0x0801.

Manufacturer IDs are allocated by NXP, customers preparing to go into production can request the allocation of a Manufacturer ID from NXP to use in their products. Customers should not use Manufacturer IDs allocated to other companies, including NXP's Manufacturer ID, in their own products.

During development the Manufacturer ID 0x0001 may be used by anyone.

2.4.1.3 Product ID (16 bits)

The least significant 16 bits are the Product ID these are allocated by the manufacturer to identify different products.

Where a manufacturer is using their own Manufacturer ID (or the global 0x0001 Manufacturer ID), in the Device ID they may allocate Product IDs as they see fit.

2.4.2 Device Type IDs (16 bits)

The 16-bit Device Type IDs are used as a short-hand to identify classes of devices. Multifunctional devices may include more than one Device Type ID. It is also valid for a device to include no Device Type IDs.

For example there may be many different manufacturers of bulbs each with a range of bulbs resulting in many different Device IDs being used in bulbs, however they may all use the standard Device Type ID of 0x00E1 to indicate a single channel dimmable bulb.

The Device Type ID is surfaced to the application layers during some communications and may be used to take different actions depending upon the Device Type ID. For example the commissioning features on the Remote Control use the Device Type ID included in a join request to determine if a device is the correct type of device that can currently be commissioned.

The Device Type IDs are made available in the DeviceID MIB present in each device.

There are two kinds of Device Type IDs:

2.4.2.1 Standard Device Type IDs

Standard Device Type IDs are allocated by NXP for use in standardised devices, these can be recognised by the most significant bit being 0.

When using a standard Device Type ID certain MIBs must be present in the device in order to provide a standardised device.

2.4.2.2 Manufacturer Device Type IDs

Manufacturer Device Type IDs can be allocated by customers using their own Manufacturer ID within the Device ID. In order to correctly determine the Device Type the Device Type ID must be used in conjunction with the Manufacturer ID within the Device ID.

2.4.3 MIB IDs (32 bits)

Each MIB has a 32-bit MIB ID which provides a convenient way to access MIBs which is independent of the order of the MIBs in a device. MIBs with the same IDs in different devices *must* contain the same set of variables. Each MIB also has a name making it easier for humans to read.



The MIB ID is often used to cache information about the variables that are present in a MIB, therefore if the variables are changed a new MIB ID should be allocated.



When changing the variables in a device during development it may be necessary to power down changed devices and reset the gateway to clear any information cached in the gateway for the changed MIB.

There are two types of MIB IDs:

2.4.3.1 Standard MIB IDs

Standard MIB IDs are allocated by NXP for use in standardised devices, these can be recognised by the upper 16 bits being set to 0xFFFF. The lower 16 bits identify the purpose of the MIB and are allocated by NXP.

Customers using standard MIB must include the variables specified for that MIB by NXP, if the variables in such a MIB are adapted by the customer the standard MIB ID should be replaced with a Manufacturer MIB ID to maintain standardisation.

2.4.3.2 Manufacturer MIB IDs

Manufacturer MIB IDs can be allocated by customers using their own Manufacturer ID within the Device ID. The upper 16 bits should be the Manufacturer ID (as used in the Device ID). The lower 16 bits identify the purpose of the MIB and are allocated by the manufacturer.

2.5 Message Transmission

There are two different ways to transmit messages in a JenNet-IP network:

2.5.1 Unicast Messaging

Unicast messages are sent to a single node. They can be used to set or get MIB variables and any response is returned using a unicast back to the requesting node. When these messages are sent they must follow the network tree and so are normally only used in a gateway network.

Typical usage is in a gateway network to monitor and control individual devices.

This method of messaging is also used in a standalone network when a remote is commissioning new devices into its network. During this time a minimal tree network is in place between the remote control and the device being commissioned, so the remote is able to use unicasts to commission devices.

2.5.2 Multicast Messaging

Multicast messages, (or broadcasts), are sent to every non-sleeping node in a network. When each node receives a multicast message it is retransmitted for other nodes to receive and forward in turn. Each node keeps a history of recently received messages allowing duplicate messages to be filtered out.

Multicast messages can be addressed to groups of devices, while each multicast is always retransmitted by every node, only nodes that are members of the group address will act upon the received message. The groups that each node is a member of can be configured using the stack's Groups MIB.

Multicast messages can be used to set MIB variables. To avoid saturating the radio bandwidth no responses are returned for received multicast messages so they cannot be used to get variables.

Typical usage is with the Remote Control broadcasting commands to control devices. Multicasts may also be issued from or via the gateway to control groups of nodes.

3 System Operation

This section describes the operation of the devices implemented in the **JenNet-IP Application Template Application Note**.

The software is written to run on the hardware included in the NXP JN516x EK001 Evaluation Kit. The Carrier Boards, Expansion Boards and JN516x modules are separately described in **JN-UG-3093 JN516x EK001 Evaluation Kit User Guide**.

The devices in this Application Note support two modes of operation:

- Gateway System in which the nodes of a WPAN can be controlled:
 - from outside the WPAN, via an IP connection from a PC
 - from within the WPAN, from other wireless devices
- Standalone System, in which the nodes of a WPAN can be controlled only from a wireless device within the WPAN (there is no IP connection)

The devices in this Application Note are capable of operating in a Standalone System however the Application Note does not include a controller device that is able to commission them into a Standalone System. (The Remote Control software in the **JN-AN-1162 JenNet-IP Smart Home Application Note** is able to commission any device into its standalone network however it does not allow the control of the devices in this Application Note.)

Therefore the only method of controlling the devices in this Application Note is via an IP connection from a PC in a Gateway System.

3.1 Gateway System Operation

This chapter describes how to use the contents of an Evaluation Kit to set up and run the JenNet-IP Application Template Application Note. This demonstration is based on a WPAN with nodes containing one of two device types:

1. Template software, which serves as example code showing how to create a basic device that joins and maintains its place in a network. This device, by its nature, does not include any additional application functionality.
2. Digital I/O software, which provides example code showing how to extend the template to include additional application functionality. This software allows the digital I/O line of the chips to be configured and controlled. The software can be remotely configured to monitor the buttons and control the LEDs on the Evaluation Kit hardware.

This chapter guides you through the set-up and operation of the demo system, as follows:

- [Gateway System Operation Overview](#) provides an overview of the demo system, describing the roles of the evaluation kit components in the demonstration
- [Setting up the Gateway System](#) details how to set up the demonstration network from the contents of the evaluation kit and run the pre-loaded software.
- [Operating the Gateway System](#) describes how to operate the demonstration.
- [Group Configuration and Control](#) describes how to form groups of devices and control them together.

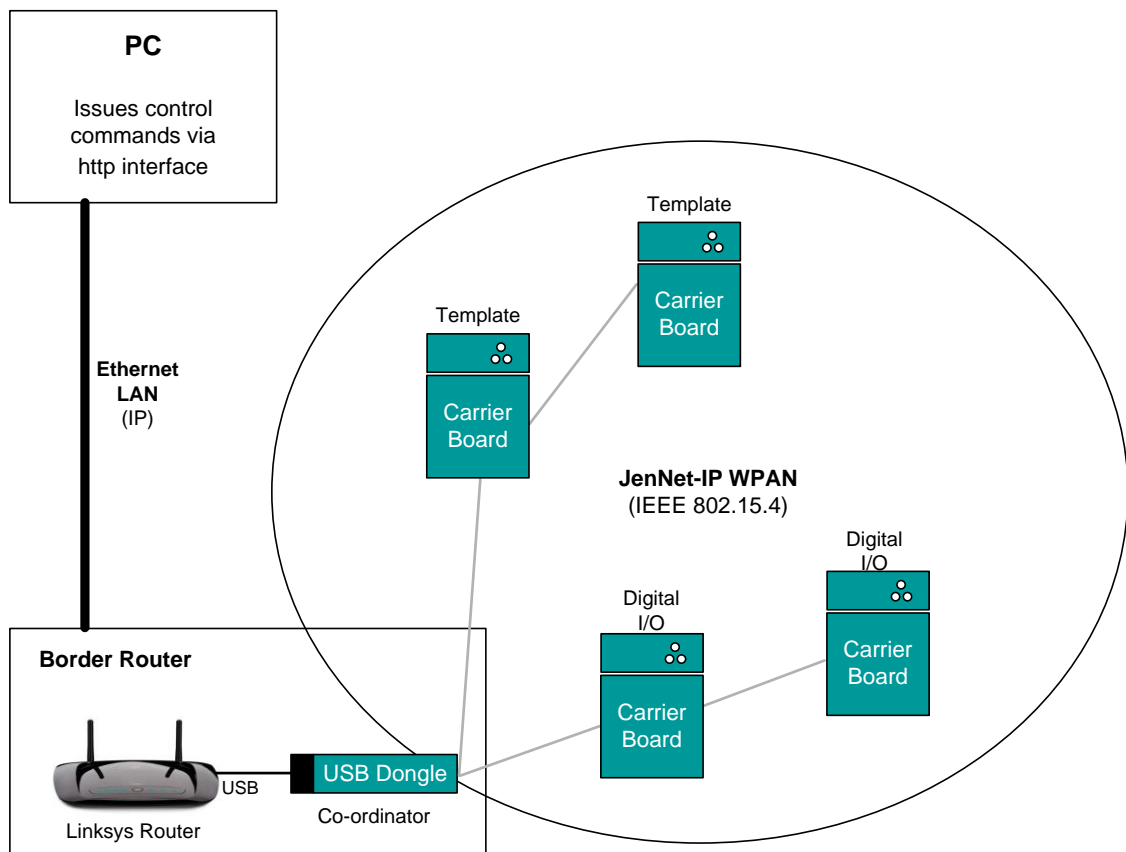
3.1.1 Gateway System Operation Overview

In the JenNet-IP Application Template a set of devices form a WPAN which can be accessed from a PC located on an Ethernet bus. The components of an Evaluation Kit are used in the demonstration as follows:

- **Carrier Boards with Generic Expansion Boards:** The four carrier boards supplied in the kit are pre-fitted with Lighting/Sensor or Generic Expansion Boards and JN516x modules. Each of these four board assemblies acts as a node of the WPAN, where the JN516x module on each node is pre-programmed as a WPAN Router. In the demonstration, the buttons and LEDs on the Carrier Board and Generic Expansion Board may be monitored and controlled when running the Digital I/O software.
- **USB Dongle:** This demonstration uses one of the supplied USB dongles programmed as a Border-Router and WPAN Co-ordinator. The dongle connects to the Linksys router (via the USB extension cable). Together they provide the Border-Router which is the interface between the WPAN and LAN/WAN domains - the dongle handles the WPAN side of this interface. The dongle is also the Co-ordinator node of the WPAN. (The second USB Dongle may be used to run the Template firmware in order to create a larger network.)
- **Linksys Router:** The Linksys router has been pre-programmed with an NXP firmware upgrade, based upon OpenWRT, which allows the router to operate in a JenNet-IP system. It is connected to the above USB dongle (via the USB extension cable). Together they provide the Border-Router which is the interface between the WPAN and LAN/WAN domains - the router handles the LAN/WAN side of the interface and connects to the Ethernet bus on which the controlling PC is located.

JenNet-IP Application Template Application Note

The Application Template system is illustrated below.



The WPAN will have a tree topology but its precise topology cannot be pre-determined since the network is formed dynamically. One or more of the Routers may be leaf-nodes of the tree, in which case their routing capability will not be used.

3.1.1.1 Device Control from a PC

In this demonstration, control and monitoring commands can be issued from a PC on an Ethernet LAN connected to the Border-Router, from where the commands will be delivered to the target nodes in a WPAN. A command can be directed to an individual node in the form of a unicast or to groups of nodes in the form of a broadcast.

A generic JenNet-IP Browser application is provided on the Linksys router that allows a PC user to monitor and control the devices in the WPAN. This application runs on the router and serves web pages to a normal web browser on the PC, allowing the user to interact with the WPAN nodes through the Border-Router.

The JenNet-IP Browser application provides a low-level interface for monitoring and controlling the devices in the WPAN, allowing the user to access the MIBs on the WPAN nodes. The application can be accessed by entering the following (case-sensitive) IP address into the web browser: <http://192.168.11.1/cgi-bin/Browser.cgi>



Note: The JenNet-IP Border-Router Configuration interface is also provided on the Linksys router. This application can be accessed by simply entering the IP address of the router into a web browser on the PC: <http://192.168.11.1/>

3.1.2 Setting Up the Gateway System

This section describes how to set up the Gateway System using the Evaluation Kit components

3.1.2.1 Programming the Device Firmware

To run the software in this Application Note the appropriate firmware must be programmed into the Evaluation Kit hardware.



Instructions on how to connect the Evaluation Kit boards to a PC and program them with firmware are included in **JN-UG-3007 JN51xx Flash Programmer User Guide**.



Pre-built firmware binaries are provided with this Application Note. If you wish to compile your own binaries instructions for importing the Application Note into Eclipse and compiling are included in **JN-UG-3064 SDK Installation and User Guide**.



The USB Dongle provided in the Evaluation Kit is used to run the Border Router firmware which is pre-loaded into the USB Dongles included in the Evaluation Kit. If the software in the USB Dongle has been changed it will need to be re-programmed with the Border Router firmware provided in **JN-AN-1110 JenNet-IP Border Router Application Note**.

Pre-built binary files are included in the Binary folder of the Application Note for programming into the Evaluation Kit Boards.

Template Device Binaries

0x11111111s_DeviceTemplate_DR1174_EndDevice_JN5168_v0000.bin
0x11111111s_DeviceTemplate_DR1174_Router_JN5168_v0000.bin

These are the binary files for the template device.

On JN516x devices the software is available to operate as either a Router or sleeping End Device node.

Digital I/O Device Binaries

0x11111111s_DeviceDio_DR1174_EndDevice_JN5168_v0000.bin
0x11111111s_DeviceDio_DR1174_Router_JN5168_v0000.bin

These are the binary files for the Digital I/O device.

On JN516x devices the software is available to operate as either a Router or sleeping End Device node.

3.1.2.2 Setting Up the LAN Part

In setting up the LAN part of the demo system, you will need the following components:

- A PC running Windows XP or Windows 7
- Linksys router and USB extension cable (from the evaluation kit)
- USB dongle (from the evaluation kit)
- Ethernet cable (from the evaluation kit)

To set up the LAN part of the system, follow the instructions below.

Step 1 Connect the PC to the Linksys router

- a) Boot up the PC.
- b) Use the supplied Ethernet cable to connect the PC to the Linksys router (but do not power on the Linksys router yet). Use a blue Ethernet socket on the router (do not use the yellow socket labelled 'Internet').

Step 2 Connect the USB dongle to the Linksys router

Connect the USB dongle (which is programmed as a Border-Router and as a WPAN Co-ordinator) to the USB socket of the Linksys router via the supplied USB extension cable (use of this cable improves the radio performance of the dongle).



Step 3 Power on the Linksys router

Connect the power supply to the Linksys router. The unit will automatically power on (this will also start the USB dongle). The power LED will first flash and then the central LED will flash. The unit is ready when the central LED stops flashing and remains illuminated.

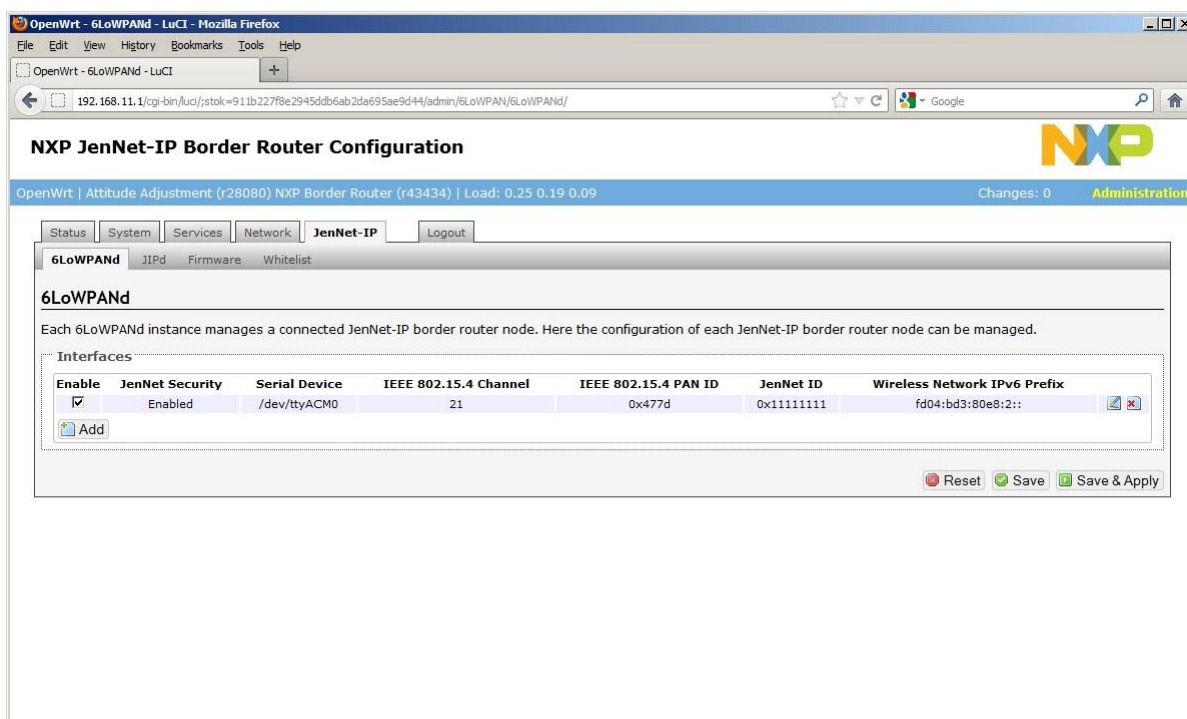
Since the USB dongle will also be the Co-ordinator of the WPAN, this device will create a network, for the moment consisting of just the Co-ordinator - the rest of the network will be formed later.

Step 4 Check the Linksys router configuration from the PC (optional)

If you wish, you can now check the system configuration on the Linksys router as described below - you should not need to change the default settings. Otherwise, continue to [Setting up the WPAN Part](#) to set up the WPAN part of the system.


- a) Launch a web browser on the PC.
- b) Access the JenNet-IP Border-Router Configuration interface on the Linksys router by entering the following IP address into the browser:
<http://192.168.11.1/>
- c) On the resulting web page, log in with username “root” and password “snap”.
- d) On the next web page, select the JenNet-IP tab, then select the 6LoWPANd sub-tab.

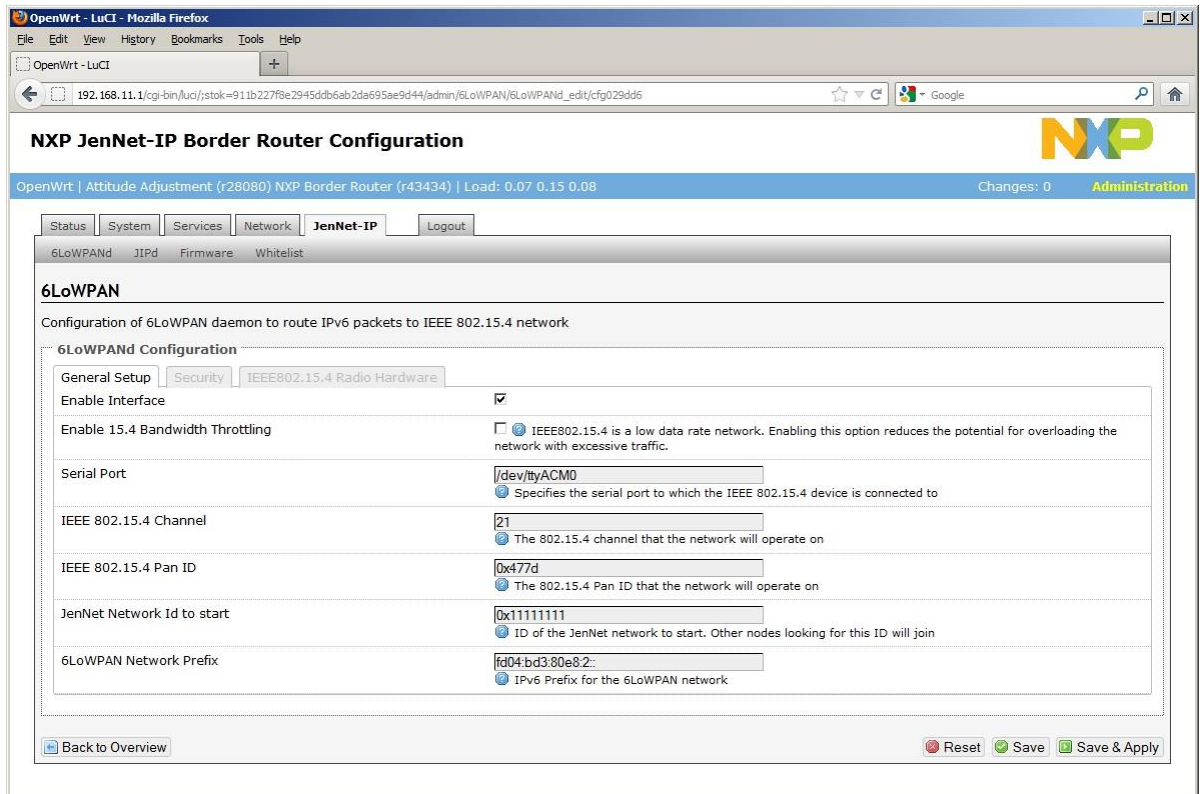
The 6LoWPANd sub-tab is illustrated in the screenshot below.



The fields in the above screenshot are described in the table below.

Field	Description
Enable	Checkbox used to enable/disable the 6LoWPANd interface
JenNet Security	Indicates whether security is enabled in the WPAN - should be enabled in this demo
Serial Device	Indicates serial port to which Border-Router node (dongle) is connected on the Linksys router
IEEE 802.15.4 Channel	Number of the radio channel used in the WPAN - selected by the Co-ordinator in this demo and should not be changed when running the demo system.
IEEE 802.15.4 PAN ID	16-bit PAN ID of wireless network – selected by the Co-ordinator in this demo and should not be changed when running the demo system
JenNet ID	32-bit Network Application ID of WPAN
Wireless Network IPv6 Pre-fix	64-bit IPv6 address prefix for WPAN

- e) In the 6LoWPANd sub-tab, click the Edit button  on the right-hand side.
- f) In the 6LoWPANd Configuration screen (which now appears), click on the General Setup tab. This displays similar fields to those listed in the table above, as shown in the screenshot below (except Security is on a separate tab).



JenNet-IP Application Template

Application Note

g) In the General Setup tab:

- Ensure that the Enable Interface checkbox is ticked.
- Ensure that the Enable 15.4 Bandwidth Throttling checkbox is unticked.
- Ensure that the JenNet Network Id to start field is set to 0x11111111 (this is an application-specific identifier).
- If required, enter a new setting for the IEEE 802.15.4 PAN ID.
- If required, enter a new setting for the 6LoWPAN Network Prefix

h) Now select the Security tab (see screenshot below) and ensure that the JenNet Security Enabled checkbox is ticked.

The screenshot shows a web browser window titled "OpenWrt - LuCI - Mozilla Firefox" displaying the "NXP JenNet-IP Border Router Configuration" page. The page has a blue header with the NXP logo and navigation tabs: Status, System, Services, Network, and JenNet-IP (selected). Below the tabs is a sub-menu: 6LoWPAN, JIPd, Firmware, and Whitelist. The main content area is titled "6LoWPAN" and contains the text "Configuration of 6LoWPAN daemon to route IPv6 packets to IEEE 802.15.4 network". Under "6LoWPANd Configuration", there are three sub-tabs: General Setup, Security (selected), and IEEE802.15.4 Radio Hardware. The Security tab contains the following fields:

- JenNet Security Enabled:** A checkbox that is checked.
- JenNet Security Key:** A text field containing "1". A tooltip below it reads: "JenNet security key. Specified like an IPv6 address e.g. 0:1:2::3:4".
- Node Authorisation Scheme:** A dropdown menu set to "RADIUS Server with PAP".
- RADIUS Server IPv6 Address:** A text field containing "fd04:bd3:80e8:1::1". A tooltip below it reads: "IPv6 Address of RADIUS server that will authorise nodes. This should usually be set to the IPv6 address of the lan interface: 'fd04:bd3:80e8:1::1'".

At the bottom of the form are three buttons: "Back to Overview", "Reset", "Save", and "Save & Apply".

i) If you have made any changes, click the Save & Apply button to implement them.

3.1.2.3 Setting Up the WPAN Part

In setting up the WPAN part of the demo system, you will need the following components:

- LAN part of the system (set up as described in [Setting Up the LAN Part](#))
- Carrier Boards fitted with JN516x modules and optional Generic (LED and Button) Expansion Boards programmed with the required firmware
- Antennae and batteries for the above boards

You can use as many of the boards as you like in this demonstration - for example, you may wish to initially use only one board.

To set up the WPAN part of the system (and therefore complete the demo system), follow the instructions below:

Step 1 Install the antenna onto a board and start the node

Perform the following for just one node:

- a) On a carrier board fitted with a JN516x module with uFL connector, install one of the supplied push-through antennae onto the board and connect the attached fly lead to the uFL connector on the module (as described in the User Guide for your Evaluation Kit).
- b) Insert four of the supplied AAA batteries onto the rear of the carrier board (the required polarities are indicated on the board). Also ensure that the batteries have been selected as the power source for the board, with the jumper J4 in the BAT position (as described in the Evaluation Kit User Guide). When battery power is supplied, the colour LED module on the lighting expansion board illuminates in red.

On power-up, the node will attempt to join the WPAN (for which the USB dongle is the Co-ordinator). There is no timeout on the node's attempt to find and join the WPAN, *but the node will not be able to join until it has been whitelisted (next step).*



Note: If the board has previously been used, it will retain settings (e.g. PAN ID) from the previous network to which it belonged. To clear this information and return to the factory settings, perform a factory reset as follows: *Wait at least 2 seconds following power-up and then press the Reset button on the carrier board 4 times with less than 2 seconds between two consecutive presses.* After the reset, the board will try to join a new network. This is illustrated in the [Node_bTestFactoryReset\(\) Functions](#) section.

JenNet-IP Application Template

Application Note

Step 5 Access the JenNet-IP Border-Router Configuration interface from the PC
If not already done (from the LAN part set-up in [Setting up the LAN Part](#)), access the JenNet-IP Border-Router Configuration interface from the PC as follows:

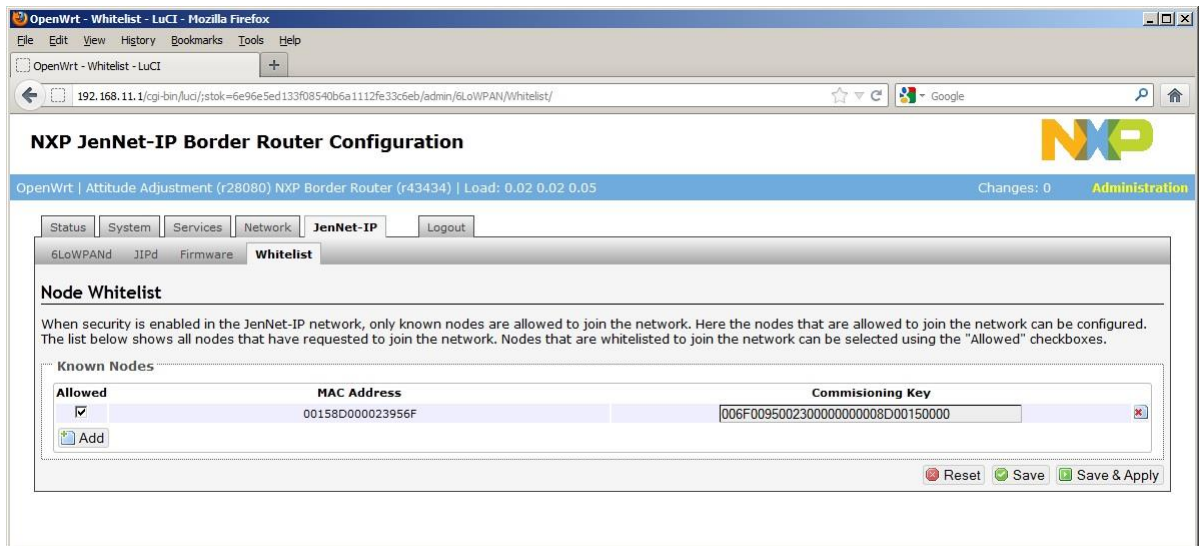
- a) Launch a web browser on the PC.
- b) Access the JenNet-IP Border-Router Configuration interface on the Linksys router by entering the following IP address into the browser:

<http://192.168.11.1/>

- c) On the resulting web page, log in with username `root` and password `snap`.

Step 6 Display the 'whitelist' of WPAN nodes in the interface on the PC

- a) In the interface, select the **JenNet-IP** tab and then select the **Whitelist** sub-tab. Normally, this sub-tab shows a list of the detected WPAN nodes, identified by their IPv6 addresses, as illustrated in the screenshot below. Those nodes that are ticked (in the checkbox on the left-hand side) are in the whitelist and so are allowed into the network. Currently, only the Evaluation Kit Board should be listed and should be unticked (greylisted) - if it does not appear, refresh the list by clicking **Whitelist** again.



- b) Put the Evaluation Kit Board into the whitelist by ticking its checkbox on the left-hand side and click the **Save & Apply** button. The unit should now be able to join the network.

- Step 7** Add additional nodes to the whitelist in the JenNet-IP Border-Router Configuration interface
- a)** Ensure that the **Whitelist** tab is shown in the JenNet-IP Border-Router Configuration interface in the web browser on the PC (see Step 3).
 - b)** Refresh the page in the browser. The new node should now appear in the list but will be unticked (greylisted). If the node fails to show in the list, you are advised to power off the board (e.g. remove the batteries) and re-start it (from **Step 4b**).
 - c)** Make a note of the node's MAC address and relate it to the physical node, for identification purposes later on (you may wish to label the node with the address).
 - d)** Put the node into the whitelist by ticking its checkbox on the left-hand side. If you have prior knowledge of the node's IEEE/MAC address and commissioning key, you can enter these details in the box revealed by clicking the **Add** button.
 - e)** Click the **Save & Apply** button. The node should now be able to join the network.

3.1.3 Operating the Gateway System

This section describes how to control the devices in the WPAN from a PC outside of the WPAN (via IP).



Note 1: Switching off the Border-Router (Linksys router and USB dongle) will cause the WPAN to automatically enter 'standalone' mode, described in [Standalone System Operation](#). Subsequently switching the Border-Router back on will cause the WPAN to return to full gateway mode.

3.1.3.1 Template Device Control from PC

The Template Devices in the WPAN can be controlled from the PC via IP. Due to its nature as a template it provides very little functionality to explore. However its MIB variables can be accessed from a PC for monitoring purposes and some variables can be written to. The template device must first be set up as described in [Setting up the Gateway System](#).

The Template Devices can be accessed from the JenNet-IP Browser, which runs on the Linksys router and is available via a normal web browser on the PC. The JenNet-IP Browser is a generic tool that can query and navigate through the devices, MIBs and variables in a JenNet-IP network.

The JenNet-IP browser is accessed by directing the web browser on the PC to the following (case-sensitive) IP address:

<http://192.168.11.1/cgi-bin/Browser.cgi>

The resulting **Network Contents** page lists the nodes in the WPAN, as illustrated in the screenshot below.

NXP JenNet-IP Browser



Network

Network Contents

Border-Router
D1199r68 32DFA0
T1174e68 32DDCE
D1199e68 32C22B
T1174r68 2FBB3C

In the above example:

- 'Border-Router' refers to the USB dongle attached to the Linksys router
- The other entries refer to the WPAN nodes that are running the template software - for example, 'D1199r68 32DFA0' where '32DFA0' is part of the IEEE/MAC address of the node

Clicking on a network node displays a MIBs page for that particular node, containing a list of the Management Information Bases (MIBs) on the node, as illustrated in the screenshot below (which shows a partial view of this page).

NXP JenNet-IP Browser



Network Node: fd04:bd3:80e8:3:215:8d00:2f:bb3c

Node "fd04:bd3:80e8:3:215:8d00:2f:bb3c" MiBs:

<u>Node</u>	MiB ID 0xffffffff00
<u>JenNet</u>	MiB ID 0xffffffff01
<u>Groups</u>	MiB ID 0xffffffff02
<u>OND</u>	MiB ID 0xffffffff03
<u>DeviceID</u>	MiB ID 0xffffffff04
<u>NodeStatus</u>	MiB ID 0xffffffe80
<u>NodeControl</u>	MiB ID 0xffffffe82
<u>NwkStatus</u>	MiB ID 0xfffffe88
<u>NwkSecurity</u>	MiB ID 0xfffffe8b

The IPv6 address of the relevant node is shown on the orange tab at the top of the page. Clicking on the adjacent **Network** tab will take you back to **Network Contents**.

JenNet-IP Application Template

Application Note

Clicking on a MIB name displays a page showing the variables within the MIB (as shown in the screenshot of the Node MIB below).

NXP JenNet-IP Browser



Network	Node: fd04:bd3:80e8:3:215:8d00:2f:bb3c	MIB: Node
MiB "Node" on Node "fd04:bd3:80e8:3:215:8d00:2f:bb3c" variables:		
MacAddr		Variable Index 0
158d00002fbb3c		
DescriptiveName		Variable Index 1
<input type="text" value="T1174r68 2FBB3C"/>		Set via Multicast Address: <input type="text"/> <input type="button" value="Update"/>
Version		Variable Index 2
1055		
TxPower		Variable Index 3
<input type="text" value="3"/>		Set via Multicast Address: <input type="text"/> <input type="button" value="Update"/>

Variable values displayed in an edit box can be written to. To change the value of the DescriptiveName variable edit the current name and click the Update button to transmit the new name to the device. This value will be saved in permanent memory and retained even if the device is power cycled, it will also be displayed in the **Network Contents** tab.

Variable values displayed as text are read only values. As the browser uses a static webpage to display its information the page must be refreshed to update any changed variables. The NwkStatus MIB contains a RunTime variable which is updated to indicate how long the device has been running, this can be observed by displaying and refreshing the NwkStatus MIB for a node in the browser.

NXP JenNet-IP Browser



Network	Node: fd04:bd3:80e8:3:215:8d00:2f:bb3c	MIB: NwkStatus
---------	--	----------------

MiB "NwkStatus" on Node "fd04:bd3:80e8:3:215:8d00:2f:bb3c" variables:

RunTime	Variable Index 0
90531	
UpCount	Variable Index 1
15	
UpTime	Variable Index 2
89934	
DownTime	Variable Index 3
597	

3.1.3.2 Digital I/O Device Configuration from PC

The Digital I/O devices in the WPAN can be configured from the PC via IP. Each of the Digital I/O lines can be configured to operate as either an input or output. When used in conjunction the Evaluation Kit Carrier and Generic Expansion Boards the LEDs can be controlled and the buttons monitored. The digital I/O device must first be set up as described in [Setting up the Gateway System](#).

The following Digital I/O lines available on the Evaluation Kit boards are shown in the table below:

Board	Usage	DIO	DIO Mask	Operation
DR1174 Carrier Board	LED D3	DIO3	0x00008	Inverted
	LED D6	DIO2	0x00004	Inverted
	Button DIO8	DIO8	0x00100	Inverted
DR1199 Generic Expansion Board	LED D1	DIO16	0x10000	Normal
	LED D2	DIO13	0x02000	Normal
	LED D3	DIO0	0x00001	Normal
	Button SW1	DIO11	0x00800	Inverted
	Button SW2	DIO12	0x01000	Inverted
	Button SW3	DIO17	0x20000	Inverted
	Button SW4	DIO1	0x00002	Inverted
DR1201 LCD Expansion Board	Button SW1	DIO11	0x00800	Inverted
	Button SW2	DIO12	0x01000	Inverted
	Button SW3	DIO17	0x20000	Inverted
	Button SW4	DIO1	0x00002	Inverted

The Operation column indicates how the DIO line is connected:

- Normal indicates that when the corresponding bit is set the LED is on or the button is pressed.
- Inverted indicates that when the corresponding bit is set the LED is off or the button is released.

To configure the pins for the LEDs and buttons on a combination of the DR1174 Carrier Board and DR1199 Generic Expansion Board, navigate to the DioConfig MIB of the device to be configured using the JenNet-IP Browser.

Enter 0x21902 into the edit box for the DirectionInput variable and click the Update button. This will configure the DIO lines connected to the buttons to operate as inputs. DIO lines not included in the mask will remain unaffected.

Enter 0x1200D into the edit box for the DirectionOutput variable and click the Update button. This will configure the DIO lines connected to the LEDs to operate as outputs. DIO lines not included in this mask will remain unaffected.

The screenshot below shows these settings:

NXP JenNet-IP Browser



MiB "DioConfig" on Node "fd04:bd3:80e8:3:215:8d00:32:dfa0" variables:

Direction	Variable Index 0
<input type="text" value="1048574"/>	Set via Multicast Address: <input type="text"/> <input type="button" value="Update"/>
Pullup	Variable Index 1
<input type="text" value="0"/>	Set via Multicast Address: <input type="text"/> <input type="button" value="Update"/>
InterruptEnabled	Variable Index 2
<input type="text" value="0"/>	Set via Multicast Address: <input type="text"/> <input type="button" value="Update"/>
InterruptEdge	Variable Index 3
<input type="text" value="0"/>	Set via Multicast Address: <input type="text"/> <input type="button" value="Update"/>
DirectionInput	Variable Index 4
<input type="text" value="0x21902"/>	Set via Multicast Address: <input type="text"/> <input type="button" value="Update"/>
DirectionOutput	Variable Index 5
<input type="text" value="0x1200D"/>	Set via Multicast Address: <input type="text"/> <input type="button" value="Update"/>
PullupEnable	Variable Index 6
<input type="text" value="0"/>	Set via Multicast Address: <input type="text"/> <input type="button" value="Update"/>
PullupDisable	Variable Index 7
<input type="text" value="0"/>	Set via Multicast Address: <input type="text"/> <input type="button" value="Update"/>
InterruptEnable	Variable Index 8
<input type="text" value="0"/>	Set via Multicast Address: <input type="text"/> <input type="button" value="Update"/>
InterruptDisable	Variable Index 9
<input type="text" value="0"/>	Set via Multicast Address: <input type="text"/> <input type="button" value="Update"/>

3.1.3.3 Digital I/O Device Monitoring from PC

The Digital I/O devices in the WPAN can be monitored from the PC via IP. Each of the Digital I/O lines configured as an input can be read. The digital I/O device must first be set up as described in [Setting up the Gateway System](#) and configured as described in [Digital I/O Device Configuration from PC](#).

To read the state of the buttons simply navigate to the DioStatus MIB page for the device to monitor. The Input variable indicates which inputs are currently set in the form of a bitmask. The value is displayed in the default decimal number base by the JenNet-IP Browser.

To check the operation of the Input variable hold down a button on the Evaluation Kit board and refresh the browser page. The value in the Input variable should change by the amount of the associated bit for the changed input. The JenNet-IP MIB Browser does not poll for new values or set up traps to monitor values making it necessary to refresh the page to read back changed values.

All buttons have inverted operation where the input is set high when the button is released and set low when the switch is pressed.

NXP JenNet-IP Browser



Network:	Node: fd04:bd3:80e8:3:215:8d00:32:dfa0	MIB: DioStatus
----------	--	----------------

MiB "DioStatus" on Node "fd04:bd3:80e8:3:215:8d00:32:dfa0" variables:

Input	Variable Index 0
137474	
Interrupt	Variable Index 1
49152	

3.1.3.1 Digital I/O Device Control from PC

The Digital I/O devices in the WPAN can be controlled from the PC via IP. Each of the Digital I/O lines configured as an output can be controlled. The digital I/O device must first be set up as described in [Setting up the Gateway System](#) and configured as described in [Digital I/O Device Configuration from PC](#).

To set the state of the LEDs navigate to the DioControl MIB page for the device to control. The Output variable indicates which outputs are currently set in the form of a bitmask. The value is displayed in the default decimal number base by the JenNet-IP Browser.

Writing to the Output variable sets the state of *all* output pins in a single operation, while this is useful in some situations it is often more useful to be able to change the state of a single (or subset) of the output pins without affecting any others. The OutputOn and OutputOff variables provide these features. The OutputOn variable will set the output state of only the pins specified in the written bitmask high leaving all other pins unchanged, similarly the OutputOff variable set the specified output pins low leaving all other pins unchanged.

The LEDs on the DR1199 Generic Expansion Boards use normal operation where setting the appropriate bit high turns on the LED. While the LEDs on the DR1174 Carrier Board use inverted operation where setting the appropriate bit low turns on the LED. Writing a value of 0x4 to the OutputOff variable will turn on LED D6 on a DR1174 Carrier Board, while writing a value of 0x4 to the OutputOn variable will turn off LED D6.

NXP JenNet-IP Browser



Network	Node: fd04:bd3:80e8:3:215:8d00:32:dfa0	MIB: DioControl
---------	--	-----------------

MiB "DioControl" on Node "fd04:bd3:80e8:3:215:8d00:32:dfa0" variables:

Output <input type="text" value="73731"/>	Variable Index 0
Set via Multicast Address: <input type="text"/>	<input type="button" value="Update"/>
OutputOn <input type="text" value="0"/>	Variable Index 1
Set via Multicast Address: <input type="text"/>	<input type="button" value="Update"/>
OutputOff <input type="text" value="0"/>	Variable Index 2
Set via Multicast Address: <input type="text"/>	<input type="button" value="Update"/>

3.1.4 Group Configuration and Control

The devices in the WPAN can be enrolled into groups. The devices within a group can be controlled synchronously by issuing a single command for the group. For example, in a real situation, the table lamps in a lounge could belong to a group, allowing all the table lamps to be switched on/off or dimmed at the same time. Note that a device can be enrolled into more than one group (or into no groups).

A group has an associated multicast address which is stored inside each member node. A command for a group includes the relevant multicast address but is broadcast to all nodes in the WPAN. A receiving node is able to use the multicast address to identify itself as a member of the group and therefore execute the command.

Device software may automatically place the device into groups on start up. The devices in this Application Note automatically place themselves into the “All Devices” group which has the IPv6 address FF15::F00F. Other device types may place themselves into groups automatically allowing all devices of a specific type to be controlled together. Devices may be removed from these predefined groups by the user so membership is not guaranteed.

IPv6 group addresses always have the most significant 16 bits set to FF15. This leaves many possible addresses available for users to use.

Some devices such as Remote Control Units need to use groups unique to that device, these take the form:

`FF15::mmmm:mmmm:mmmm:mmmm:gggg`

Where

The *mmmm* components are the MAC address of the device

The *gggg* bits are sequentially allocated by the device so it may have many groups it can transmit to.

By incorporating the MAC address of the transmitting device into the group address it reduces the likelihood of a device being accidentally added to a group that can be controlled by another device in an unintended way. Such devices often have methods to place devices within direct range into and out of their groups.

When creating groups from outside the WPAN similar techniques can be used to ensure that group addresses remain unique to the device transmitting commands to the group.

When transmitting a command to a group it takes the form of a write to a MIB variable. For this to be effective the MIB and variable must be present in the receiving device in addition to the device being a member of the addressed group.

Groups can be configured from the remote PC and, where enabled, from other devices within the WPAN such as a Remote Control Unit. Group configuration is described below.

3.1.4.1 Configuring Groups on the PC

This section describes how to set up groups of devices (WPAN nodes) for control from a PC via an IP connection. Groups of devices can be set up from the JenNet-IP Browser, which runs on the Linksys router and is accessed via a normal web browser on the PC.

The JenNet-IP browser is accessed by directing the web browser on the PC to the following (case-sensitive) IP address:

<http://192.168.11.1/cgi-bin/Browser.cgi>

To configure the node's group memberships, first navigate to the page for the node that is to be added to the group. Click on the **Groups** MIB on the Node's MIB page. This takes you to the **Groups** MIB page that lists the variables contained in the Groups MIB, as illustrated in the screenshot below.

NXP JenNet-IP Browser



Network	Node: fd04:bd3:80e8:3:215:8d00:32:d88c	MIB: Groups
---------	--	-------------

MiB "Groups" on Node "fd04:bd3:80e8:3:215:8d00:32:d88c" variables:

Groups Variable Index 0

000 { 0x15f00f }

AddGroup Variable Index 1

Set via Multicast Address:

RemoveGroup Variable Index 2

Set via Multicast Address:

ClearGroups Variable Index 3

Set via Multicast Address:

The current group addresses the node is a member of are displayed in the Groups list at the top of the page.

Note that the leading FF of the group address is omitted from the display and does not need to be included when manipulating the other variables, this part of the address is assumed to be FF.

Also note that the group addresses are entered as a single string of hexadecimal digits, (following the leading 0x). The leading value of 15 is assumed to be in the most significant position of the address, all following digits are shifted down to the least significant positions.

So an IPv6 group address FF15::F00F is displayed and should be entered as "0x15f00f".

This page can be used to modify the group memberships of the node (with IPv6 address indicated at the top of the page), by means of the following fields:

- **AddGroup:** To add the node to a group:
 - a) Enter the identifier of the group in this field.
 - b) Click on the **Update** button for AddGroup.
 - c) Click on the orange **MIB Groups** tab to refresh the page. The new group should now appear in the **Groups** section of the page.
- **RemoveGroup:** To remove the node from a group:
 - a) Enter the identifier of the group in this field.
 - b) Click on the **Update** button for RemoveGroup.
 - c) Click on the orange **MIB Groups** tab to refresh the page. The group should now disappear from the **Groups** section of the page.
- **ClearGroups:** To remove the node from all groups:
 - a) Enter any non-zero value in this field.
 - b) Click on the **Update** button for ClearGroups.
 - c) Click on the orange **MIB Groups** tab to refresh the page. All groups should now disappear from the **Groups** section of the page.

3.1.4.2 Controlling Groups of Devices from the PC

This section describes how to control groups of devices (WPAN nodes) from a PC via an IP connection. Groups of devices can be controlled from the JenNet-IP Browser, which runs on the Linksys router and is accessed via a normal web browser on the PC.

The JenNet-IP browser is accessed by directing the web browser on the PC to the following (case-sensitive) IP address:

<http://192.168.11.1/cgi-bin/Browser.cgi>

To control a number of devices a command is broadcast that performs a write to a MIB variable. For the command to be effective the nodes must be members of the group plus the MIB and variable must be present in the device.

Navigate to the MIB containing the variable to be edited on any device that includes the MIB in the network. Normal updates are unicast only to the node being displayed, however entering a group address into the Set via Multicast Address field will override that behaviour and transmit the command as a multicast to the specified address when the Update button is pressed. This is shown in the screen shot below:

NXP JenNet-IP Browser



Network	Node: fd04:bd3:80e8:3:215:8d00:32:d88c	MIB: NodeControl
MiB "NodeControl" on Node "fd04:bd3:80e8:3:215:8d00:32:d88c" variables:		
Variable Index 0		
Reset	Set via Multicast Address: ff15::f00f	Update
10		
Variable Index 1		
FactoryReset	Set via Multicast Address:	Update
0		

The example above shows the contents of the NodeControl MIB. Pressing the Update button will transmit a command to start a 10 second reset timer to any device that is a member of the "All Devices" group (address FF15::F00F), all devices are automatically added to this group on start-up. When the reset timer reaches zero all nodes will be reset.

4 MIB Variable Reference

This section provides reference information on the MIBs and variables created by the **JenNet-IP Application Template Application Note**.

The application MIBs are grouped together logically into functional groups and each individual MIB contains a logically grouped set of variables.

Note that the JenNet-IP stack provides a number of MIBs and variables that become available in every JenNet-IP device, these MIBs and variables are documented in **JN-UG-3080 JenNet-IP WPAN Stack User Guide**. The MIBs implemented by the stack are the Node, JenNet, Groups, OND and DeviceID MIBs.

4.1 Node MIBs

The Node MIBs provide access to functionality relating to node operation.
These MIBs are implemented in the MibCommon library.

4.1.1 NodeStatus MIB (0xFFFFFE80)

The NodeStatus MIB provides status information for the node.

4.1.1.1 SystemStatus Variable

Description

The SystemStatus variable provides access to the System Status register as set at power-on or reset.

Storage

Permanent

Type

Uint16 Unsigned Integer, 16 bits

Access

Read

Values

<i>0x0001</i>	Wake-up Status – if set the device has woken from sleep.
<i>0x0002</i>	Memory Status – indicates woken from sleep with memory held.
<i>0x0004</i>	Analogue Peripheral Power Status – set when the voltage regulator for the analogue peripheral is enabled.
<i>0x0008</i>	Protocol clock Status – set when protocol clock is enabled and running with no significant lag.
<i>0x0010</i>	MISOS – value on the SPI MISOS pin.
<i>0x0020</i>	Voltage Brownout Status – set when the supply voltage is below the VBO threshold.
<i>0x0040</i>	Voltage Brownout Status Valid – set when the VBO status bit is valid.
<i>0x0080</i>	Watchdog Reset Status – set when the watchdog has reset the node.
<i>0x0100</i>	Voltage Brownout Reset Status – set when the VBO has reset the node.

Default

Determined at power-on or reset.

Trap Notifications

None.

4.1.1.2 ColdStartCount Variable

Description

The ColdStartCount variable provides a count of the number of cold starts (power-ons or resets) experienced by the device.

Storage

Permanent

Type

Uint16 Unsigned Integer, 16 bits

Access

Read

Values

0 to 65535

Default

0

Trap Notifications

None.

4.1.1.3 ResetCount Variable

Description

The ResetCount variable provides a count of the number of deliberate resets applied to the device (by writing to the NodeControl MIB Reset variable).

Storage

Permanent

Type

Uint16 Unsigned Integer, 16 bits

Access

Read

Values

0 to 65535

Default

0

Trap Notifications

None.

4.1.1.4 WatchdogCount Variable

Description

The WatchdogCount variable provides a count of the number resets caused by the watchdog tripping.

Storage

Permanent

Type

Uint16 Unsigned Integer, 16 bits

Access

Read

Values

0 to 65535

Default

0

Trap Notifications

None.

4.1.1.5 BrownoutCount Variable

Description

The BrownoutCount variable provides a count of the number resets caused by the brownout tripping.

Storage

Permanent

Type

Uint16 Unsigned Integer, 16 bits

Access

Read

Values

0 to 65535

Default

0

Trap Notifications

None.

4.1.1.6 HeapMin Variable

Description

The HeapMin variable indicates the start address of the heap of dynamically allocated memory, this shouldn't change at run-time. It effectively indicates the end of the statically allocated software image and data in RAM and so reflects the RAM requirements of the application.

Storage

Volatile

Type

UInt32 Unsigned Integer, 32 bits

Access

Read

Values

0x04000000 – 0x04FFFFFF

Default

Depends upon application size

Trap Notifications

None.

4.1.1.7 HeapMax Variable

Description

The HeapMax variable indicates the maximum address memory that has been dynamically allocated on the heap. The difference between this value and HeapMin represents the maximum size the heap has reached while the software has been running.

Storage

Volatile

Type

UInt32 Unsigned Integer, 32 bits

Access

Read

Values

0x04000000 – 0x04FFFFFF

Default

Depends upon application size

Trap Notifications

None.

4.1.1.8 StackMin Variable

Description

The StackMin variable indicates the minimum address in memory that has been reached by the processor's stack. The stack grows down from address 0x04FFFFFF so the difference indicates the maximum size the stack has reached.

The amount of unused memory can be calculated by the difference between StackMin and HeapMax if this is zero or very low the application runs the risk of crashing due to the stack or heap overwriting each other.



The method currently used to monitor this value does not detect the allocation of data to the stack which is not written to. The current application appears to do this so values reported here may not be accurate.

Storage

Volatile

Type

UInt32 Unsigned Integer, 32 bits

Access

Read

Values

0x04000000 – 0x04FFFFFF

Default

Depends upon application functions

Trap Notifications

None.

4.1.2 NodeConfig MIB (0xFFFFFE81)

The NodeConfig MIB is reserved for future use and is currently not present in devices.

4.1.3 NodeControl MIB (0xFFFFFE82)

The NodeControl MIB provides control over the node's operation.

4.1.3.1 Reset Variable

Description

The Reset variable causes the device to countdown for a number of seconds and then perform a software reset.

Storage

Volatile

Type

Uint16 Unsigned Integer, 16 bits

Access

Read, Write

Values

<i>0</i>	A reset is not pending, a pending reset can be cancelled by writing 0 into this variable.
<i>1 to 65535</i>	Number of seconds remaining until the node resets.

Default

0

Trap Notifications

On remote edits.

4.1.3.2 FactoryReset Variable

Description

The FactoryReset variable causes the device to countdown for a number of seconds and then perform a factory reset returning all settings to their default values.

Storage

Volatile

Type

Uint16 Unsigned Integer, 16 bits

Access

Read, Write

Values

<i>0</i>	A factory reset is not pending, a pending reset can be cancelled by writing 0 into this variable.
<i>1 to 65535</i>	Number of seconds remaining until the node resets.

Default

0

Trap Notifications

On remote edits.

4.2 Network MIBs

The Network MIBs provide variables to monitor and configure the device within the network.

4.2.1 NwkConfig MIB (0xFFFFFE89)

The NwkConfig MIB contains variables that can be used to configure the network settings used by the node. To apply the settings the node must be reset or power cycled.

This MIB is not normally registered with the stack making its variables inaccessible through the network. This results in a hardcoded set of configuration values, though normally these variables would not need to be changed at run-time.

4.2.1.1 DeviceType

Description

The DeviceType variable specifies which type of network device the device runs as. By default devices run as Router devices, some devices may be built to run as End Devices.

Devices may be configured to run as a Coordinator device for test or demo purposes to allow gateway-less operation.



To re-configure a Coordinator device back to a Router device a mechanism must be provided on another device in the network or it will need to be re-programmed due to the lack of a gateway providing general access.

Storage

Permanent

Type

Uint8 Unsigned Integer, 8 bits

Access

Read, Write

Values

1	Coordinator.
2	Router

Default

2	Router.
---	---------

Trap Notifications

On remote edits.

4.2.1.2 NetworkId

Description

The NetworkId variable specifies the 32-bit Network ID used to form or join a JenNet-IP network.



When changing this variable care should be taken to ensure there is a suitable network to join. If unable to join a network it will not be possible to update the variable to a new value.

Storage

Permanent

Type

Uint32 Unsigned Integer, 32 bits

Access

Read, Write

Values

0x00000001 to 0xfffffffffe

Default

0x11111111

Trap Notifications

On remote edits.

4.2.1.3 ScanChannels

Description

The ScanChannels variable specifies the 32-bit Scan Channels bitmask used to select the possible channels when forming or joining a JenNet-IP network.



When changing this variable care should be taken to ensure there is a suitable network to join. If unable to join a network it will not be possible to update the variable to a new value.

Storage

Permanent

Type

Uint32 Unsigned Integer, 32 bits

Access

Read, Write

Values

0x00000800 to 0x07fff800

Default

0x07fff800 All channels.

Trap Notifications

On remote edits.

4.2.1.4 PanId

Description

The PanId variable specifies the PAN ID of the network the device should try to join, a value of 0xffff allows any network to be joined.

Storage

Permanent

Type

Uint16 Unsigned Integer, 16 bits

Access

Read, Write

Values

<i>0x0000 to 0xfffe</i>	Join specified PAN ID only
<i>0xffff</i>	Join any PAN ID

Default

0xffff

Trap Notifications

On remote edits.

4.2.1.5 EndDeviceActivityTimeout

Description

The EndDeviceActivityTimeout variable specifies the period between messages from child End Devices before it is considered lost. Set in units of 100ms.

Storage

Permanent

Type

Uint32 Unsigned Integer, 32 bits

Access

Read, Write

Values

0 to 4294967296

Default

600 (60 seconds)

Trap Notifications

On remote edits.

4.2.1.6 FrameCounterDelta

Description

The FrameCounterDelta variable specifies how far security frame counters need to advance before they are written to flash in devices in a standalone system.

Storage

Permanent

Type

Uint16 Unsigned Integer, 16 bits

Access

Read, Write

Values

10 to 65535

Default

4096

Trap Notifications

On remote edits.

4.2.1.7 StackModelInit

Description

The StackModelInit variable specifies the initial stack mode when creating or joining a network.

Storage

Permanent

Type

Uint16 Unsigned Integer, 16 bits

Access

Read, Write

Values

<i>0</i>	Gateway mode
<i>1</i>	Standalone mode

Default

<i>0</i>	Gateway Mode
----------	--------------

Trap Notifications

On remote edits.

4.2.2 NwkProfile MIB (0xFFFFFE8D)

The NwkProfile MIB contains variables that can be used to configure the network profile settings of the device. It also provides an alternative parent selection algorithm that may be used by devices trying to join the network making use of a preferred LQI threshold.

While the source code for this MIB is included in the Application Note it is not normally compiled into applications but is made available for testing and evaluation use. When this MIB is compiled it is always compiled into the application it is not part of the MibCommon library.

4.2.2.1 MaxFailedPackets Variable

Description

This variable specifies the maximum number of failed packets before a parent is considered lost.

Storage

Permanent

Type

UInt8 Unsigned Integer, 8 bits

Access

Read, Write

Values

0 to 255

Default

7

Trap Notifications

On remote writes.

4.2.2.2 MinBeaconLqi Variable

Description

This variable specifies the minimum acceptable LQI level for beacon responses from potential parents. Nodes with a weaker signal than this won't be considered as a parent for joining the network.

Storage

Permanent

Type

UInt8 Unsigned Integer, 8 bits

Access

Read, Write

Values

0 to 255

Default

18

Trap Notifications

On remote writes.

4.2.2.3 PrfBeaconLqi Variable

Description

This variable specifies a preferred LQI for beacon responses from potential parents. Nodes with a higher LQI than this are preferred to nodes with a lower LQI. Where there are a number of potential parents above this level the standard Depth, Children then LQI ordering is used. For weaker responses below the preferred LQI the ordering is reversed to LQI, Children then Depth thus favouring the strongest response from a set of weak responses. This alternative ordering algorithm is part of the application and is not part of the stack.

Storage

Permanent

Type

UInt8 Unsigned Integer, 8 bits

Access

Read, Write

Values

0 to 255

Default

25

Trap Notifications

On remote writes.

4.2.2.4 RouterPingPeriod Variable

Description

This variable specifies the period of automatic pings a node sends to its parent in the network in units of 10ms.

Storage

Permanent

Type

Uint16 Unsigned Integer, 16 bits

Access

Read, Write

Values

0 to 65535

Default

1500

Trap Notifications

On remote writes.

4.2.3 NwkStatus MIB (0xFFFFFE88)

The NwkStatus MIB contains variables that indicate the status of the device within the network.

4.2.3.1 RunTime Variable

Description

The variable specifies how long the software has been running in seconds.

Storage

Volatile – but derived from UpTime and DownTime variables

Type

Uint32 Unsigned Integer, 32 bits

Access

Read

Values

0 to 4294967296

Default

0

Trap Notifications

Every hour the device has run.
Upon joining a network.

4.2.3.2 UpCount Variable

Description

The variable specifies how many times the device has joined or re-joined the network.

Storage

Permanent

Type

Uint16 Unsigned Integer, 16 bits

Access

Read

Values

0 to 65535

Default

0

Trap Notifications

Upon joining a network.

4.2.3.3 UpTime Variables

Description

The variable specifies how long the device has been in the network in seconds.

Storage

Permanent

Type

Uint32 Unsigned Integer, 32 bits

Access

Read

Values

0 to 4294967296

Default

0

Trap Notifications

Every hour the device has been in the network.
Upon joining a network.

4.2.3.4 DownTime Variables

Description

The variable specifies how long the device software has been out of the network in seconds.

Storage

Permanent

Type

Uint32 Unsigned Integer, 32 bits

Access

Read

Values

0 to 4294967296

Default

0

Trap Notifications

Upon joining a network.

4.2.4 NwkControl MIB (0xFFFFFE8A)

The NwkControl MIB is reserved for future use to contain variables that allow to network operation of the device to be controlled.

4.2.5 NwkSecurity MIB (0xFFFFFE8B)

The NwkSecurity MIB contains the various security keys used by the device and also information on the network.

4.2.5.1 KeyNetwork

Description

The KeyNetwork variable contains the key used by the device while in the network, the key is usually obtained when joining the network.

Storage

Permanent

Type

Blob Blob, 128 bits

Access

Read, Write

Values

Any Network key.

Default

Unspecified

Trap Notifications

On remote edits.

4.2.5.2 KeyGateway

Description

The KeyGateway variable contains the commissioning key used by the device while trying to join a gateway network.

Storage

Permanent

Type

Blob Blob, 128 bits

Access

Read, Write

Values

Any Gateway commissioning key.

Default

Unspecified

Trap Notifications

On remote edits.

4.2.5.3 KeyStandalone

Description

The KeyStandalone variable contains the commissioning key used by the device while trying to join a standalone network.

Storage

Permanent

Type

Blob Blob, 128 bits

Access

Read, Write

Values

Any Standalone commissioning key.

Default

Unspecified

Trap Notifications

On remote edits.

4.2.5.4 Channel

Description

The Channel variable contains the radio channel the device is currently operating on.

Storage

Volatile

Type

uint8 Unsigned integer 8 bits

Access

Read

Values

11-26 Current channel

Default

Unspecified

Trap Notifications

None

4.2.5.5 PanId

Description

The PanId variable contains the PAN ID of the network radio channel the device is currently a member of.

Storage

Volatile

Type

Uint16 Unsigned integer 16 bits

Access

Read

Values

Any Current PAN ID

Default

Unspecified

Trap Notifications

None

4.2.5.6 Rejoin Variable

Description

The Rejoin variable causes the device to countdown for a number of seconds then discard the current network channel, PAN ID and Network Key and perform a reset to force a full re-join of the network. This is useful when moving a network to a new channel.

Storage

Volatile

Type

Uint16 Unsigned Integer, 16 bits

Access

Read, Write

Values

<i>0</i>	A re-join is not pending, a pending re-join can be cancelled by writing 0 into this variable.
<i>1 to 65535</i>	Number of seconds remaining until the node re-joins.

Default

0

Trap Notifications

On remote edits.

4.2.6 NwkTest MIB (0xFFFFFE8C)

The NwkTest MIB contains variables that can be used to run packet error and signal strength tests.

While the source code for this MIB is included in the Application Note it is not normally compiled into applications but is made available for testing and evaluation use. This MIB is always compiled into the application, it is not compiled into the MibCommon library.

4.2.6.1 Tests Variable

Description

The Tests variable is used to initiate a number of transmission tests. When set to a non-zero value the node will transmit the specified number of packets to its parent to measure the packet error rate and signal strength of the returned packets.

Storage

Volatile

Type

Uint8 Unsigned Integer, 8 bits

Access

Read, Write

Values

0	No effect.
1 to 255	Number of test transmissions.

Default

0

Trap Notifications

On remote edits.

4.2.6.2 TxReq Variable

Description

The TxReq variable contains the number of test transmission attempts made.

Storage

Volatile

Type

UInt8 Unsigned Integer, 8 bits

Access

Read

Values

0 to 255 Number of test transmission requests.

Default

0

Trap Notifications

On remote edits.

4.2.6.3 TxOk Variable

Description

The TxOk variable contains the number of successful test transmissions.

Storage

Volatile

Type

UInt8 Unsigned Integer, 8 bits

Access

Read

Values

0 to 255 Number of successful test transmissions.

Default

0

Trap Notifications

On remote edits.

4.2.6.4 RxOk Variable

Description

The RxOk variable contains the number of successfully received test transmission responses.

Storage

Volatile

Type

UInt8 Unsigned Integer, 8 bits

Access

Read

Values

0 to 255 Number of successful test responses.

Default

0

Trap Notifications

On remote edits.

4.2.6.5 RxLqiMin Variable

Description

The RxLqiMin variable contains the lowest LQI from the successfully received test transmission responses.

Storage

Volatile

Type

UInt8 Unsigned Integer, 8 bits

Access

Read

Values

0 to 255 Minimum LQI of test responses.

Default

0

Trap Notifications

On remote edits.

4.2.6.6 RxLqiMax Variable

Description

The RxLqiMax variable contains the highest LQI from the successfully received test transmission responses.

Storage

Volatile

Type

UInt8 Unsigned Integer, 8 bits

Access

Read

Values

0 to 255 Maximum LQI of test responses.

Default

0

Trap Notifications

On remote edits.

4.2.6.7 RxLqiMean Variable

Description

The RxLqiMean variable contains the mean average LQI from the successfully received test transmission responses.

Storage

Volatile

Type

UInt8 Unsigned Integer, 8 bits

Access

Read

Values

0 to 255 Mean average LQI of test responses.

Default

0

Trap Notifications

On remote edits.

4.2.6.8 CwChannel Variable

Description

The CwChannel is a placeholder variable to allow constant wave transmission on the specified channel. The source code to enable this is not included in the public Application Note.

Storage

Volatile

Type

UInt8 Unsigned Integer, 8 bits

Access

Read, Write

Values

11 to 26 Channel for constant wave transmission.

Default

0

Trap Notifications

On remote edits.

4.2.6.9 MacRetries Variable

Description

The MacRetries variable specifies the maximum number of MAC level retries that should be used when transmitting test packets.

Storage

Volatile

Type

UInt8 Unsigned Integer, 8 bits

Access

Read, Write

Values

0 to 255 Maximum MAC level retries for test packets.

Default

0

Trap Notifications

On remote edits.

4.2.6.10 TxLqiMin Variable

Description

The TxLqiMin variable contains the lowest LQI from the successfully transmitted test transmission requests. This is only filled in when the parent device also has the NwkTest MIB implemented.

Storage

Volatile

Type

Uint8 Unsigned Integer, 8 bits

Access

Read

Values

0 to 255 Minimum LQI of test transmissions.

Default

0

Trap Notifications

On remote edits.

4.2.6.11 TxLqiMax Variable

Description

The TxLqiMax variable contains the highest LQI from the successfully transmitted test transmission requests. This is only filled in when the parent device also has the NwkTest MIB implemented.

Storage

Volatile

Type

Uint8 Unsigned Integer, 8 bits

Access

Read

Values

0 to 255 Maximum LQI of test transmissions.

Default

0

Trap Notifications

On remote edits.

4.2.6.12 TxLqiMean Variable

Description

The TxLqiMean variable contains the mean average LQI from the successfully transmitted test transmission requests. This is only filled in when the parent device also has the NwkTest MIB implemented.

Storage

Volatile

Type

Uint8 Unsigned Integer, 8 bits

Access

Read

Values

0 to 255 Mean average LQI of test transmissions.

Default

0

Trap Notifications

On remote edits.

4.2.6.13 RxLqi Variable

Description

The RxLqi variable contains the LQI of the last received packet on the node. When reading this variable it should be the LQI of the last hop of the get request itself. This test transmissions actually request this variable which allows the transmitted LQIs to be calculated from the returned values.

Storage

Volatile

Type

Uint8 Unsigned Integer, 8 bits

Access

Read

Values

0 to 255 LQI of last received packet.

Default

0

Trap Notifications

On remote edits.

4.3 Peripheral MIBs

The Peripheral MIBs provide generic information for peripheral devices.

The AdcStatus MIB is part of the MibCommon library.

The DIO MIBs are formed from the DIO MIB modules, they are not part of the MibCommon library.

4.3.1 AdcStatus MIB (0xFFFFFE90)

The AdcStatus MIB contains variables that indicate the status of the ADC inputs. This module provides useful functionality that can be enabled and used without the MIB and variables being registered.

Each variable and its use is described in the following chapters:

4.3.1.1 Mask Variable

Description

The variable indicates which ADC inputs have been enabled by the application.

Storage

Volatile

Type

Uint8 Unsigned Integer, 8 bits

Access

Read

Values

<i>0x01</i>	ADC 1
<i>0x02</i>	ADC 2
<i>0x04</i>	ADC 3
<i>0x08</i>	ADC 4
<i>0x10</i>	ADC On-Chip Temperature, when this is enabled this module will automatically perform temperature calibration, and push and pull the oscillator if an oscillator control pin is specified.
<i>0x20</i>	ADC On-Chip Voltage

Default

Set by application

Trap Notifications

None

4.3.1.2 Read Table

Description

The Read table provides access to the current raw readings from the 6 ADC inputs. Usually other MIBs will pick up these readings and convert them to the appropriate measurement.

Storage

Volatile

Type

Uint16 [6] Unsigned Integer, 16 bits, 6 entries

Access

Read

Values

0 to 65535 ADC reading.

Default

0

Trap Notifications

None.

4.3.1.3 ChipTemp Variable

Description

The ChipTemp variable contains the on-chip temperature in 10ths of a degree Centigrade when the ADC source is being read.

Storage

Volatile

Type

Int16 Signed Integer, 16 bits

Access

Read

Values

0 to 65535 On-chip temperature ADC reading.

Default

0

Trap Notifications

None.

4.3.1.4 CalTemp Variable

Description

The CalTemp variable contains the on-chip temperature in 10ths of a degree Centigrade when the radio was last calibrated.

Storage

Volatile

Type

Int16 Signed Integer, 16 bits

Access

Read

Values

0 to 65535 On-chip temperature ADC reading at time of last radio calibration.

Default

0

Trap Notifications

None.

4.3.1.5 Oscillator Variable

Description

The Oscillator variable contains the level of oscillator pulling currently being applied.

Storage

Volatile

Type

Uint8 Unsigned Integer, 8 bits

Access

Read

Values

0 to 255 Level of oscillator pulling being applied.

Default

0

Trap Notifications

None.

4.3.2 DioStatus MIB (0xFFFFFE70)

The DioStatus MIB contains variables that indicate the status of the Digital I/O input lines.

All these variables operate as bitmaps where each DIO is represented by a bit (bit 4 corresponds to DIO4). To manipulate the DIOs the appropriate variable bits should be manipulated.

Each variable and its use is described in the following chapters:

4.3.2.1 Input Variable

Description

The Input variable indicates which inputs are high or low, where a bit is set the corresponding input is high.

The value returned matches the value returned by the `u32AHB_DioReadInput()` function in the Integrated Peripherals API.

Storage

Volatile

Type

UInt32 Unsigned Integer, 32 bits

Access

Read

Values

0 to 0xFFFFFFFF Bitmap where set bits indicate the corresponding DIOs are high.

Default

0

Trap Notifications

Upon input DIO changes.

4.3.2.2 Interrupt Variable

Description

The Interrupt variable indicates which input(s) last generated an interrupt where a bit is set the corresponding input generated the most recent interrupt.

The value returned matches the value returned by the `u32AHI_DioInterruptStatus()` function in the Integrated Peripherals API.

Storage

Volatile

Type

UInt32 Unsigned Integer, 32 bits

Access

Read

Values

0 to 0xFFFFFFFF Bitmap where set bits indicate the corresponding DIOs generated the most recent interrupt.

Default

0

Trap Notifications

Upon input DIO interrupts.

4.3.3 DioConfig MIB (0xFFFFFE71)

The DioConfig MIB contains variables that allow the configuration of the DIO lines to be read or altered.

All these variables operate as bitmaps where each DIO is represented by a bit (bit 4 corresponds to DIO4). To manipulate the DIOs the appropriate variable bits should be manipulated.

Each variable and its use is described in the following chapters:

4.3.3.1 Direction Variable

Description

The Direction variable indicates which DIO lines are inputs or outputs.

When written to it configures the direction of all DIO lines in a single operation. The DirectionInput and DirectionOutput MIB variables may be used to alter the direction of individual or subsets of the DIO lines.

This variable mirrors the REG_GPIO_DIR register.

Storage

Permanent

Type

Uint32 Unsigned Integer, 32 bits

Access

Read, Write

Values

0 to 0xFFFFFFFF Bitmap where set bits indicate the corresponding DIOs are outputs or clear bits indicate an input.

Default

0

Trap Notifications

Upon input DIO direction changes.

4.3.3.2 Pullup Variable

Description

The Pullup variable indicates which DIO lines have the internal pull-up resistor enabled.

When written to it configures the pull-ups of all DIO lines in a single operation. The PullupEnable and PullupDisable MIB variables may be used to alter the pull-ups for individual or subsets of the DIO lines.

This variable mirrors the REG_SYS_PULLUP register DIO bits.

Storage

Permanent

Type

Uint32 Unsigned Integer, 32 bits

Access

Read, Write

Values

0 to 0xFFFFFFFF Bitmap where set bits indicate the corresponding DIOs pull-ups are enabled.

Default

0

Trap Notifications

Upon input DIO pull-up changes.

4.3.3.3 InterruptEnabled Variable

Description

The InterruptEnabled variable indicates which DIO input lines are enabled to generate interrupts.

When written to it configures the interrupts of all DIO input lines in a single operation. The InterruptEnable and InterruptDisable MIB variables may be used to configure the interrupts for individual or subsets of the DIO input lines.

This variable mirrors the REG_SYS_WK_EM register DIO bits.

Storage

Permanent

Type

Uint32 Unsigned Integer, 32 bits

Access

Read, Write

Values

0 to 0xFFFFFFFF Bitmap where set bits indicate the corresponding DIOs input should generate interrupts.

Default

0

Trap Notifications

Upon input DIO enabled interrupt changes.

4.3.3.4 InterruptEdge Variable

Description

The InterruptEdge variable indicates upon which edge DIO input lines should generate interrupts.

When written to it configures the interrupt edge of all DIO input lines in a single operation. The InterruptRising and InterruptFalling MIB variables may be used to configure the interrupt edge for individual or subsets of the DIO input lines.

This variable mirrors the REG_SYS_WK_ET register DIO bits.

Storage

Permanent

Type

Uint32 Unsigned Integer, 32 bits

Access

Read, Write

Values

0 to 0xFFFFFFFF Bitmap where set bits indicate the corresponding DIOs input should generate interrupts.

Default

0

Trap Notifications

Upon input DIO interrupt edge changes.

4.3.3.5 DirectionInput Variable

Description

The DirectionInput configures individual or a subset of the DIO lines to operate as inputs.

Writing to this variable is the equivalent of the u32Inputs parameter when calling the vAHB_DioSetDirection() function.

Storage

Volatile (but stored as part of the Direction variable)

Type

Uint32 Unsigned Integer, 32 bits

Access

Write

Values

0 to 0xFFFFFFFF Bitmap where set bits indicate the corresponding DIOs should be configured as inputs. The direction of DIOs corresponding to clear bits are not affected.

Default

0

Trap Notifications

None

4.3.3.6 DirectionOutput Variable

Description

The DirectionOutput configures individual or a subset of the DIO lines to operate as outputs.

Writing to this variable is the equivalent of the u32Outputs parameter when calling the vAH1_DioSetDirection() function.

Storage

Volatile (but stored as part of the Direction variable)

Type

Uint32 Unsigned Integer, 32 bits

Access

Write

Values

0 to 0xFFFFFFFF Bitmap where set bits indicate the corresponding DIOs should be configured as outputs. The direction of DIOs corresponding to clear bits are not affected.

Default

0

Trap Notifications

None

4.3.3.7 PullupEnable Variable

Description

The PullupEnable enables an individual or subset of the DIO internal pull-up resistors.

Writing to this variable is the equivalent of the u32On parameter when calling the vAHI_DioSetPullup() function.

Storage

Volatile (but stored as part of the Pullup variable)

Type

Uint32 Unsigned Integer, 32 bits

Access

Write

Values

0 to 0xFFFFFFFF Bitmap where set bits indicate the corresponding DIOs pull-ups should be enabled. The pull-ups of DIOs corresponding to clear bits are not affected.

Default

0

Trap Notifications

None

4.3.3.8 PullupDisable Variable

Description

The PullupDisable disables an individual or subset of the DIO internal pull-up resistors.

Writing to this variable is the equivalent of the u32Off parameter when calling the vAHI_DioSetPullup() function.

Storage

Volatile (but stored as part of the Pullup variable)

Type

UInt32 Unsigned Integer, 32 bits

Access

Write

Values

0 to 0xFFFFFFFF Bitmap where set bits indicate the corresponding DIOs pull-ups should be disabled. The pull-ups of DIOs corresponding to clear bits are not affected.

Default

0

Trap Notifications

None

4.3.3.9 InterruptEnable Variable

Description

The InterruptEnable enables an individual or subset of the DIO inputs to generate interrupts.

Writing to this variable is the equivalent of the u32Enable parameter when calling the vAHB_DioInterruptEnable() function.

Storage

Volatile (but stored as part of the InterruptEnabled variable)

Type

UInt32 Unsigned Integer, 32 bits

Access

Write

Values

0 to 0xFFFFFFFF Bitmap where set bits indicate the corresponding DIO input interrupts should be enabled. The interrupt configuration of DIOs corresponding to clear bits are not affected.

Default

0

Trap Notifications

None

4.3.3.10 InterruptDisable Variable

Description

The InterruptDisable disables an individual or subset of the DIO inputs to generate interrupts.

Writing to this variable is the equivalent of the u32Disable parameter when calling the vAHB_DioInterruptEnable() function.

Storage

Volatile (but stored as part of the InterruptEnabled variable)

Type

UInt32 Unsigned Integer, 32 bits

Access

Write

Values

0 to 0xFFFFFFFF Bitmap where set bits indicate the corresponding DIO input interrupts should be disabled. The interrupt configuration of DIOs corresponding to clear bits are not affected.

Default

0

Trap Notifications

None

4.3.3.11 InterruptRising Variable

Description

The InterruptRising configures an individual or subset of the DIO inputs to generate interrupts on a rising edge.

Writing to this variable is the equivalent of the u32Rising parameter when calling the vAHI_DioInterruptEdge() function.

Storage

Volatile (but stored as part of the InterruptEdge variable)

Type

UInt32 Unsigned Integer, 32 bits

Access

Write

Values

0 to 0xFFFFFFFF Bitmap where set bits indicate the corresponding DIO input interrupts should be raised on a rising edge. The interrupt configuration of DIOs corresponding to clear bits are not affected.

Default

0

Trap Notifications

None

4.3.3.12 InterruptFalling Variable

Description

The InterruptFalling configures an individual or subset of the DIO inputs to generate interrupts on a falling edge.

Writing to this variable is the equivalent of the u32Falling parameter when calling the vAHB_DioInterruptEdge() function.

Storage

Volatile (but stored as part of the InterruptEdge variable)

Type

UInt32 Unsigned Integer, 32 bits

Access

Write

Values

0 to 0xFFFFFFFF Bitmap where set bits indicate the corresponding DIO input interrupts should be raised on a falling edge. The interrupt configuration of DIOs corresponding to clear bits are not affected.

Default

0

Trap Notifications

None

4.3.4 DioControl MIB (0xFFFFFE72)

The DioControl MIB contains variables that allow the DIO output lines to be controlled.

All these variables operate as bitmaps where each DIO is represented by a bit (bit 4 corresponds to DIO4). To manipulate the DIOs the appropriate variable bits should be manipulated.

Each variable and its use is described in the following chapters:

4.3.4.1 Output Variable

Description

The Output variable indicates which DIO output lines are high or low.

When written to it sets the state of all DIO output lines in a single operation. The OutputOn and OutputOff MIB variables may be used to set the state of individual or subsets of the DIO output lines.

This variable mirrors the REG_GPIO_DOUT register.

Storage

Permanent

Type

Uint32 Unsigned Integer, 32 bits

Access

Read, Write

Values

0 to 0xFFFFFFFF Bitmap where set bits indicate the corresponding DIO outputs are high and clear bits are low.

Default

0

Trap Notifications

Upon input DIO output changes.

4.3.4.2 OutputOn Variable

Description

The OutputOn variable sets individual or a subset of the DIO output line states to high.

Writing to this variable is the equivalent of the u32On parameter when calling the vAHI_DioSetOutput() function.

Storage

Volatile (but stored as part of the Output variable)

Type

UInt32 Unsigned Integer, 32 bits

Access

Write

Values

0 to 0xFFFFFFFF Bitmap where set bits indicate the corresponding DIO outputs should be high. The state of DIO outputs corresponding to clear bits are not affected.

Default

0

Trap Notifications

None

4.3.4.3 OutputOff Variable

Description

The OutputOff variable sets individual or a subset of the DIO output line states to low.

Writing to this variable is the equivalent of the u32Off parameter when calling the vAHI_DioSetOutput() function.

Storage

Volatile (but stored as part of the Output variable)

Type

UInt32 Unsigned Integer, 32 bits

Access

Write

Values

0 to 0xFFFFFFFF Bitmap where set bits indicate the corresponding DIO outputs should be low. The state of DIO outputs corresponding to clear bits are not affected.

Default

0

Trap Notifications

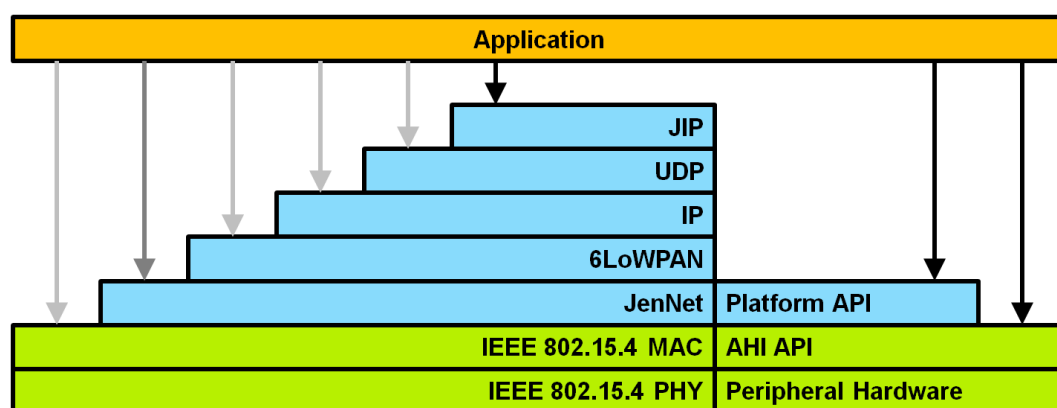
None

5 Software Reference

This section provides information on the software of each of the device types. Every device follows the same basic flow of function calls that form a JenNet-IP device application as detailed in **JN-UG-3080 JenNet-IP WPAN Stack User Guide**.

JenNet-IP applications are built using the JenNet-IP WPAN Stack API and Integrated Peripherals API. The majority of application calls are into the top layers of each stack, though there may be cases where the lower layers are accessed directly.

The diagram below shows the layers upon which the application is written. The black arrows represent the majority of calls to the upper layers of the stack, while the lighter grey arrows indicate the minority of calls into the lower layers:



5.1 DeviceTemplate

The **DeviceTemplate** folder of the Application Note contains source code for a template device. The template device can operate as a Router or End Device node. It is able to join a network and maintain its place within the network but does not provide any other functionality.

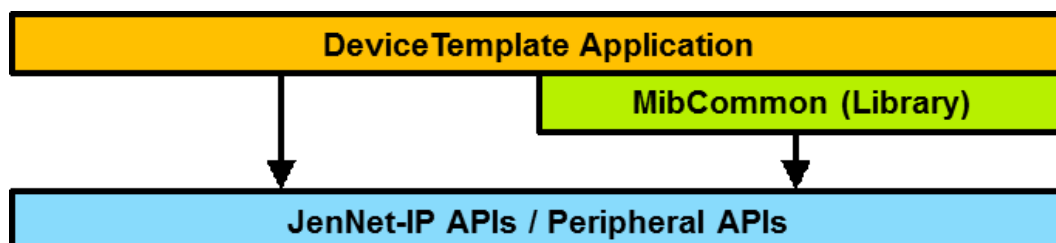
Adding code to the template device is the best way to create new types of JenNet-IP devices, the developer can focus on writing code to provide the device's functionality using the network maintenance code unchanged.

The common source **Common\Node.c** actually provides the majority of the network joining and maintenance code. **Node.c** in turn makes use of a number of MIBs that can be used to monitor, configure and control the individual node and its operation within the network, these MIBs are implemented in the **MibCommon** Library.

The source code in the **DeviceTemplate** folder contains the main module implementing the standard JIP call-back functions, which then make calls into the stack and MIB libraries as required.

On JN5142-J01 chips the MibCommon library is within the chip's ROM. In some places patch functions are called instead of library functions to alter the behaviour programmed into the ROM. The libraries used on the JN5148-J01 and JN516x chips contain identical code to that used in the JN5142-J01 ROM and so patch functions are used in exactly the same way.

The following diagram shows the layers that form the DeviceTemplate application on top of the JenNet-IP WPAN Stack:



5.1.1 Makefile

The **makefile** (or values passed into it on the command line) determines which CPU and hardware platform the software is built to run upon.

Makefile variables are also used to specify network parameters and settings.

Further makefile variables control which MIBs are built into the application and also which MIBs are registered with JIP to make them available for use in the device, this is most useful to add test MIBs during development while reducing the memory overhead.

The following sections describe the significant variables used in the makefile.

5.1.1.1 JENNIC_CHIP

This variable specifies the microcontroller the template software should be compiled for.

When compiling from Eclipse this value is passed into make on the command line depending upon which build target was selected.

Each individual value creates a separate **#define** that is used to determine the microcontroller type where necessary in the source code.

The following values are valid:

JN5168 for JN5168-001 chips, this creates the **#define JENNIC_CHIP_JN5168**.

JN5164 for JN5164-001 chips, this creates the **#define JENNIC_CHIP_JN5164**.

5.1.1.2 DEVICE_NAME

This variable specifies the hardware platform the template software should be compiled for, this generally equates to the circuit board the microcontroller is mounted upon, but may also take into account additional hardware on other circuit boards within the device.

In some devices, such as bulbs, the same source may be recompiled for different hardware with just minor changes, usually at the lowest hardware driver level, to support different hardware platforms.

When compiling in Eclipse this value is passed into make on the command line depending upon which build target was selected.

The **#define MK_DEVICE_NAME** is set to the contents of this variable in the form of a string to allow source code decisions to be made based upon the hardware platform being targeted.

The template device supports only the following value:

DR1174 for the following DR1174 Evaluation Kit Base Board

This variable is also used to select the values for the JIP Device ID, JIP Device Type and Over Network Download Image via the **JIP_DEVICE_ID**, **JIP_DEVICE_TYPE** and **OND_DEVICE_TYPE** makefile variables. The first two of these are exposed to the source files as the **#defines MK_JIP_DEVICE_ID** and **MK_JIP_DEVICE_TYPE**.

5.1.1.3 NODE_TYPE

This variable specifies which node type the application should be compiled for. The following options are available:

Router, the node runs as a Router device. Routers must be permanently powered and can accept child devices to extend the coverage of the network.

EndDevice, the node runs as an End Device. End Devices may be battery powered and can sleep and wake in order to preserve power. However they are unable to accept children of their own. Only JN516x chips may operate as End Devices.

The node type is exposed to the application source files as the #define **MK_NODE_TYPE**.

5.1.1.4 NETWORK_ID

This variable specifies the 32-bit JenNet-IP Network ID used to control which network the template binary is able to join. This value is surfaced to the source code via the #define **MK_NETWORK_ID**.

Where devices are being created with the intention of interoperating with other standard JenNet-IP devices and networks the default value of 0x11111111 should be retained.

Where manufacturers are creating closed systems built from only their products a value may be chosen at random.

5.1.1.5 CHANNEL

This variable specifies the channels that the device may operate on. The default value of 0 allows all channels, a value between 11 and 26 allows only that single channel. The value is surfaced to the source code via the #define **MK_CHANNEL**

5.1.1.6 SECURITY

This variable specifies if radio communications should be encrypted. The default value of 1 enables security mode, while a value of 0 disables security mode. This value is surfaced to the source code via the #define **MK_SECURITY**.

It is recommended that security be used, while the application can be compiled for unsecured use it is not a supported mode of operation for the application.

5.1.1.7 PRODUCTION

This variable specifies if the binary is a production build and is surfaced to the source code via the #define **MK_PRODUCTION**.

A value of 1 enables a production build, this will override the **SECURITY** variable value by enforcing encryption of radio data, and also reset the software if an exception is raised.

The default value of 0 disables a production build.

5.1.1.8 OND_CHIPSET

This variable specifies the chip the binary is built for as used by the Over Network Download (OND) functionality.

5.1.1.9 OND_DEVICE_TYPE

This variable specifies a 32-bit identifier for the binary image as used by the Over Network Download (OND) functionality.

In most cases this matches the **JIP_DEVICE_ID** but some differ for historical reasons.

5.1.1.10 TRACE

When set to 1 this value enables debugging of application events to the UART. This adds considerable extra code and is unlikely to operate on JN5142-J01 microcontrollers without removing other features to make room for the debugging routines and messages.

The default value of 0 disables the debug build.

5.1.1.11 JIP_CR_MANUFACTURER_ID

This is the 16-bit Manufacturer ID that forms part of the 32-bit JIP Device ID.

This value is used for Coordinator or Router node types with the most significant bit cleared to indicate a non-sleeping device.

5.1.1.12 JIP_ED_MANUFACTURER_ID

This is the 16-bit Manufacturer ID that forms part of the 32-bit JIP Device ID.

This value is used for End Device node types with the most significant bit set to indicate a sleeping device.

5.1.1.13 JIP_PRODUCT_ID

This is the 16-bit Product ID that forms part of the 32-bit JIP Device ID.

5.1.1.14 JIP_DEVICE_ID

This is the 32-bit JIP Device ID formed from the Manufacturer and Product IDs.

This value is used to identify different devices within a JenNet-IP network.

5.1.1.15 JIP_NODE_NAME

This variable specifies the default value for the Node MIB's DescriptiveName variable.

The default value forms a name from shortened forms of the Device Type, Product ID, Node Type and Chip variables.

This variable may be overridden from the command line.

This variable is surfaced to the application via the #define **MK_JIP_NODE_NAME**.

The software appends the least significant 3 bytes of the device's MAC address in hexadecimal to complete the name.

5.1.1.16 MIB Build Flags

The set of variables beginning **BLD_MIB** determine which MIBs are compiled into the application. A value of 1 will build that a MIB into the application, while a value of 0 will exclude it.

These flags are used in the makefile to specify compilation of appropriate source files.

The flags are also used in the source code via the equivalent #defines beginning **MK_BLD_MIB**.

Removing unnecessary MIBs from a device during development frees up additional memory that may allow the use of other test MIBs and UART debugging code.

5.1.1.17 MIB Registration Flags

The set of variables beginning **REG_MIB** determine which MIBs are registered with the stack making their variables available for remote access. The corresponding **BLD_MIB** variable must be 1 for this flag to have any effect.

When set to 1 the MIB will be registered and the variables available for remote access, when set to 0 the MIB is not registered.

These flags are used by the source code via the equivalent #defines beginning **MK_REG_MIB**.

Compiling a MIB but leaving it unregistered allows the MIB to perform its role in the device using a set of effectively hardcoded values while freeing up RAM to be used for other purposes.

The NwkConfig MIB is built into the application but the MIB is not registered by default. It is unlikely to be necessary to change network configuration values at run-time but they still need to be applied in order to get the network up and running in the first place. If access at run-time were required during development the MIB could be registered by altering the appropriate **REG_MIB** variable value.

5.1.1.18 VERSION

This variable embeds a 16-bit version number into the binary file that can be queried remotely and also used to automate software downloads using the Over Network Download features of JenNet-IP.

This value may be passed in on the command line. Pre-built binaries in the Application Note will have their version number set via the command line, each release will increase this value from its previous value.

Building without passing in a value will cause a value to be automatically generated from the day of the week, the hour and minute of the build. While this will produce different values for each build the counter will effectively reset to a lower value every 7 days.

The automatic Over Network Download features rely upon an increasing version number to be effective, a formal release scheme that includes increasing version numbers is recommended to make best use of this feature.

5.1.1.19 Binary File Naming

The names of the binary files incorporate a number of the variables described above in the following format:

**{NETWORK_ID}{SECURITY_CHAR}_CH{CHANNEL}_DeviceTemplate_
{DEVICE_NAME}_{NODE_TYPE}_{JENNIC_CHIP}_{BUILD}_v{VERSION}.bin**

Where:

{NETWORK_ID} is the **NETWORK_ID** variable value.

{SECURITY_CHAR} is *p* for a production build, *s* for a secure build, *u* for an unsecure build.

{CHANNEL} specifies the single channel the device will operate on. If all channels are supported this component is not included in the name.

{DEVICE_NAME} is the **DEVICE_NAME** variable value.

{NODE_TYPE} is the **NODE_TYPE** variable value.

{JENNIC_CHIP} is the **JENNIC_CHIP** variable value.

{BUILD} is set to *DEBUG* where the **TRACE** variable is 1 and is omitted from the filename for non-debug binaries.

{VERSION} is the value of the **VERSION** variable, this is only included in the filename when specified on the command line to make.

The compilation produces a single file for JN516x devices:

.bin may be used both when directly programming a device using one of the Flash Programmer utilities and also updating devices using the OND mechanism.

5.1.2 DeviceDefs.h

This header file contains a few #defines that can be used to configure the default behaviour of the device.

The initial set of operating defines are also used by the **CommonNode.c** module and so must be present for all devices.

A set of debug flags are then included that can be used to configure which types of debug to output when debugging is enabled.

5.1.3 DeviceTemplate.c

DeviceTemplate.c contains the main source code for the template application.

The standard JIP call-back functions are implemented in this source file along with code to operate the application at the highest level. However the main body of code that performs the actual work is mostly contained in the common modules and MIB libraries used by the application. As such it can be used as a template for constructing other device types by implementing and calling into a different set of MIBs as required.

The following sections describe the features of the **DeviceTemplate.c** source code. For functions called during initialisation of the device they are mostly presented in the order in which they are called, though it is not a fully linear sequence.

5.1.3.1 #defines

There are a number of local #define values in **DeviceTemplate.c** that control the operation of the device. The most notable are described below:

#define DEVICE_ADC_MASK

This value defines the mask of ADC readings that should be monitored by the AdcStatus MIB.

The on-chip temperature sensor should be included in order to allow recalibration of the radio and oscillator control due to changes in temperature.

Some hardware platforms use an ADC input to monitor the bus voltage in the device which may need to be above a particular level to allow operation of the device.

The value is therefore selected from a combination of the hardware platform and microcontroller being used.

#define DEVICE_ADC_SRC_BUS_VOLTS

This value determines which of the ADC inputs is being used to monitor the device's bus voltage so the appropriate reading can be passed to other modules for monitoring.

The value is selected from a combination of the hardware platform and microcontroller being used.

#define DEVICE_ADC_PERIOD 25

This is the period at which the ADC readings are made in 10ms intervals, the default value of 25 equates to 250ms so each reading is taken 4 times per second.

#define DEVICE_DIO_OSC_PULL 16

This specifies the digital output that is to be used to push or pull the oscillator to compensate for temperature changes on JN514x chips.

5.1.3.2 External Data

A number of externally declared variables are used by **DeviceTemplate.c**.

Each MIB built into the application requires two variables to be accessed in **DeviceTemplate.c**:

- Data structure, data used by the MIB is contained in a structure of the type **tsMibName** with the variable name **sMibName**, (where *Name* is the actual name of the MIB.) These data structure types are defined in the corresponding **MibName.h** include file of the Application Note, while the structure itself is declared in **the MibNameDec.c** source file of the application note which contains the MIB declaration.
- MIB handle, passed to JIP functions to allow access to MIBs and variables. These are of the type **thJIP_Mib** and named **hMibName** (where *Name* is the actual name of the MIB). The variable it actually declared in the **MibNameDec.c** source file of the application note.

5.1.3.3 AppColdStart() Function

```
void AppColdStart ( void );
```

This function is the entry point to the application following a reset or waking from sleep without memory held.

It simply calls **Device_vlnit()**.

5.1.3.4 AppWarmStart() Function

```
void AppwarmStart ( void );
```

This function is the entry point to the application following a wake from sleep with memory held, which should only happen on sleeping End Devices.

It simply calls **Device_vlnit()**.

5.1.3.5 Device_vInit() Function

```
void Device_vInit ( bool_t bWarmStart );
```

This function controls the overall initialisation of the device.

The code mostly consists of calling initialisation functions in various other stack, peripheral and library modules, to ensure they are ready to be used. These include the exception handlers and UART debugging modules.

The common node handling module is initialised by a call to **Node_vInit()**.

The next initialisation steps for a cold start are:

Call one of the **Node_bTestFactoryReset()** functions to test if a factory reset should be applied due to an on – off – on – off – on sequence.

A call is made to **Device_vPdmInit()** to initialise the Persistent Data Manager and data used by the MIBs in the application.

If a factory reset is required the **Device_vReset()** function is called to carry out the reset.

A call is made to **Device_eJiplInit()** which takes care of initialising the JenNet-IP stack and begin the process of joining a network.

For a warm start, (an End Device waking from sleep):

The stack function **i6LP_ResumeStack()** is called to resume the stack after the warm start.

Once the above initialisation is completed the software enters the main loop, contained in the **Device_vMain()** function.

If **Device_vMain()** is allowed to exit on an End Device the node is placed into sleep mode with a call to **Device_vSleep()**.

5.1.3.6 Device_vPdmInit() Function

```
void Device_vPdmInit ( void );
```

This function simply calls **Node_vPdmInit()** to initialise the PDM and each of the common MIBs used by the application.

When building upon the template additional MIBs may be initialised *following* the call to **Node_vPdmInit()** once the PDM has been initialised.

5.1.3.7 Device_vReset() Function

```
void Device_vReset ( bool_t bFactoryReset );
```

This function is used to reset the device, the parameter determines whether it should be a standard reset or a factory reset.

In the template this function simply calls the common **Node_vReset()** function which resets data in the common MIBs, (if appropriate for a factory reset), before resetting the device.

When building upon the template additional MIBs may be factory reset *before* the call to **Node_vReset()** where the device is actually reset.

5.1.3.8 Device_eJipInit() Function

```
teJIP_Status Device_eJipInit ( void );
```

This function initialises the JIP stack and registers the common MIBs with the stack by calling the common **Node_eJipInit()** function.

When building upon the template additional MIBs may be registered with the stack *after* the call to **Node_eJipInit()** when the stack is up and running.

5.1.3.9 v6LP_ConfigureNetwork() Function

```
void v6LP_ConfigureNetwork (
    tsNetworkConfigData *psNetworkConfigData );
```

This call-back function is called by the stack from the **eJIP_Init()** function during initialisation to allow the operation of the stack to be configured.

This function simply calls the common **Node_v6lpConfigureNetwork()** function to handle this task.

5.1.3.10 Device_vMain() Function

```
void Device_vMain ( void );
```

This function contains the main application loop which runs while the device is to stay awake.

Each time around the loop the on-chip watchdog is restarted, the tick timer events are handled with a call to **Device_vTick()**, then the node is placed into doze mode to preserve power until the next interrupt is raised.

This function only returns when the software decides that the device should be placed into a sleep mode.

5.1.3.11 v6LP_DataEvent() Function

```
void v6LP_DataEvent ( int          iSocket,
                      te6LP_DataEvent eEvent,
                      ts6LP_SockAddr *psAddr,
                      uint8          u8AddrLen );
```

This call-back function is called by the stack for data events at the 6LowPAN level this function simply calls the common **Node_v6lpDataEvent()** function.

As this application is written to operate at the JIP level, (reading and writing to MIB variables), any packets received from this level of the stack are simply discarded by **Node_v6lpDataEvent()**.

5.1.3.12 vJIP_StackEvent() Function

```
void vJIP_StackEvent ( te6LP_StackEvent    eEvent,
                      uint8                *pu8Data,
                      uint8                u8DataLen );
```

This call-back function is used to inform the application of stack events relating to the status of the device in the network. This function simply calls the common **Node_bJipStackEvent()** function to handle these events.

The return value from **Node_bJipStackEvent()** indicates if an End Device poll has indicated that there is no data remaining in the parent device. In the application template End Devices are readied for sleep mode when this takes place.

When building upon the template stack events may be passed to other modules from this function as required.

5.1.3.13 v6LP_PeripheralEvent() Function

```
void v6LP_PeripheralEvent ( uint32 u32Device,
                           uint32 u32ItemBitmap );
```

This call-back function is called by the stack each time a peripheral raises an interrupt. This function is called from within the interrupt context. The following peripherals are handled in this function:

E_AHI_DEVICE_TICK_TIMER

The JenNet-IP stack runs the tick timer so it raises an interrupt every 10ms. This is used internally by JenNet-IP for timing and may also be used by applications so long as its operation is unchanged.

As the function is running in the interrupt context the application simply increments a counter each time the interrupt takes place in order to minimize processing while in the interrupt handler. The function **Device_vTick()** which is called each time around the main loop reads the counter and manages the timing tasks within the application.

E_AHI_DEVICE_ANALOGUE

When using the common **Node.c** software it configures the AdcStatus MIB to manage the ADC peripherals to take regular readings. Each time a reading is completed an interrupt will be raised. The interrupts are passed on to **Node.c** by calling the **Node_u8Analogue()** function.

The **Node_u8Analogue()** function returns the ADC input for the completed reading. The ADC readings can be passed into other MIBs for further processing when building upon the template.

Interrupts from other peripherals can be accessed here when adapting the template to create other device types.

5.1.3.14 Device_vTick() Function

```
void Device_vTick ( void );
```

This function is called each time around the main loop its main purpose is to handle the timing of the application.

First the counter that is incremented in **v6LP_PeripheralEvent()** is checked to see if a tick timer interrupt has been generated. A further nested counter is used to monitor the passing of 1 second intervals within the application.

The tick event is then passed into the common **Node.c** module by calling the **Node_vTick()** function, this allows the node module to perform its timing tasks.

When extending the template with additional functionality other timing tasks may be performed here.

Finally the stack **v6LP_Tick()** function is called once whenever the tick timer has interrupted. This function allows the stack to perform its internal processing and timing functions and takes care of running the radio system.

5.1.3.15 Device_vException() Function

```
void Device_vException ( uint32 u32HeapAddr,  
                        uint32 u32Vector,  
                        uint32 u32Code );
```

This function, if present in an application, is called following the standard exception handler in **Exception.c**. It may be used to take additional actions if an exception is raised.

In the template the software is simply restarted.

5.1.3.16 vJIP_StayAwakeRequest() Function

```
void vJIP_StayAwakeRequest ( void );
```

This function is called in End Device nodes if the stack receives a request to stay awake. Some applications may set this flag in request messages when they need to make a number of requests of an End Device in an attempt to keep it awake.

The awake timer **u32Awake** is simply set to the value of the #define **DEVICE_ED_REQ_AWAKE** (default value is 500ms). The Template application will not allow the device to sleep while this timer is running.

5.1.3.17 Device_vSleep() Function

```
void Device_vSleep ( void );
```

This function is called in End Device nodes if the main loop is allowed to exit and controls placing the node into sleep mode.

The function **Node_vSleep()** is called to allow the common pre-sleep handling to take place.

When building upon the template additional pre-sleep handling can be added here.

The device is then placed into sleep mode with a call to **v6LP_Sleep()**.

Finally the device busy loops until the stack places the device into sleep mode.

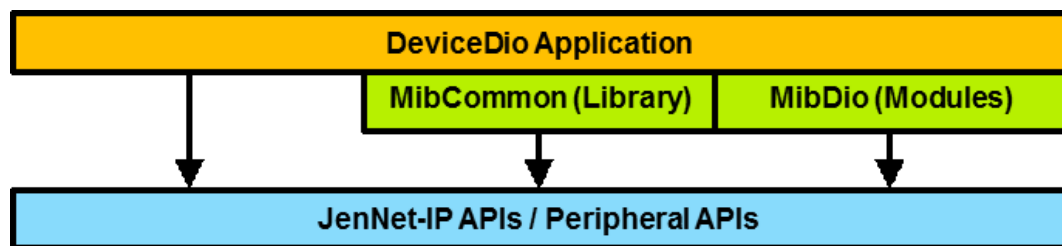
5.2 DeviceDio

The **DeviceDio** folder of the Application Note contains source code for a generic Digital Input/Output (DIO) device. This allows the DIO pins of the microcontroller to be configured, monitored and controlled. The DIO device can operate as a Router or End Device node.

The source code has been constructed by adding to the code for the template device described in [DeviceTemplate](#). Most of the additional functionality is contained within the DIO MIB source code described in DIO MIB. The code in the **DeviceDio** folder just makes calls to the DIO MIB functions as appropriate. The descriptions of the source code in this section cover only the additional code added to the template source code.

Adding additional code to the template device is the best way to create new types of JenNet-IP devices, the developer can focus on writing code to provide the device's functionality using the network maintenance code as-is.

The following diagram shows the layers that form the DeviceDio application on top of the JenNet-IP WPAN Stack:



5.2.1 Makefile

The **makefile** (or values passed into it on the command line) determines which CPU and hardware platform the software is built to run upon.

Makefile variables are also used to specify network parameters and settings.

Further makefile variables control which MIBs are built into the application and also which MIBs are registered with JIP to make them available for use in the device, this is most useful to add test MIBs during development while reducing the memory overhead.

The **makefile** is almost identical to the template device **makefile** described in the [DeviceDio Makefile](#) section. Additional source code modules are included to add the code for the DIO MIBs. The following sections describe the changed variables used in the DIO device makefile.

5.2.1.1 DEVICE_NAME

The DIO device supports only the following value:

DR1199 for the DR1199 Evaluation Kit Generic Shield.

This board is selected as it supports a number of switches that may be configured for use as digital inputs and LEDs that may be configured for use as digital outputs. However as the software provides generic access to the DIO pins it may be run on any hardware platform.

5.2.1.2 MIB Build Flags

The DIO Device makefile adds additional variables to control the building of the DioStatus, DioConfig and DioControl MIBs.

5.2.1.3 MIB Registration Flags

The DIO Device makefile adds additional variables to control the registration of the DioStatus, DioConfig and DioControl MIBs.

5.2.1.4 Binary File Naming

The names of the binary files incorporate a number of the makefile variables in the following format:

**{NETWORK_ID}{SECURITY_CHAR}_CH{CHANNEL}_DeviceDio_
{DEVICE_NAME}_{NODE_TYPE}_{JENNIC_CHIP}_{BUILD}_v{VERSION}.bin**

Where:

{NETWORK_ID} is the **NETWORK_ID** variable value.

{SECURITY_CHAR} is *p* for a production build, *s* for a secure build, *u* for an unsecure build.

{CHANNEL} specifies the single channel the device will operate on. If all channels are supported this component is not included in the name.

{DEVICE_NAME} is the **DEVICE_NAME** variable value.

{NODE_TYPE} is the **NODE_TYPE** variable value.

{JENNIC_CHIP} is the **JENNIC_CHIP** variable value.

{BUILD} is set to *DEBUG* where the **TRACE** variable is 1 and is omitted from the filename for non-debug binaries.

{VERSION} is the value of the **VERSION** variable, this is only included in the filename when specified on the command line to make.

The compilation produces a single file for JN516x devices:

.bin may be used both when directly programming a device using one of the Flash Programmer utilities and also updating devices using the OND mechanism.

5.2.2 DeviceDefs.h

This header file contains a few #defines that can be used to configure the default behaviour of the device.

The #defines used here are the same as those used in the template device.

5.2.3 DeviceDio.c

DeviceDio.c contains the main source code for the Digital IO application.

The standard JIP call-back functions are implemented in this source file along with code to operate the application at the highest level. However the main body of code that performs the actual work is mostly contained in the MIB libraries used by the application.

The **DeviceDio.c** file is based upon the **DeviceTemplate.c** source file. The main differences being the use of the DIO MIB modules that implement the DIO functionality. The following sections briefly describe the features of the **DeviceDio.c** source that differ from those in the **DeviceTemplate.c** source code.

5.2.3.1 #include

Additional #includes are used to provide access to the DIO MIB modules used in **DeviceDio.c**.

5.2.3.2 External Data

External data variables are added to access the data and handles of the DIO MIBs.

5.2.3.3 Device_vPdmInit() Function

```
void Device_vPdmInit ( void );
```

The DIO MIB modules are initialised here.

5.2.3.4 Device_vReset() Function

```
void Device_vReset ( bool_t bFactoryReset );
```

The factory reset of the DIO MIBs permanent data is performed here.

5.2.3.5 Device_eJipInit() Function

```
teJIP_Status Device_eJipInit ( void );
```

The DIO MIBs are registered with the JenNet-IP stack in this function.

5.2.3.6 v6LP_PeripheralEvent() Function

```
void v6LP_PeripheralEvent ( uint32 u32Device,  
                           uint32 u32ItemBitmap );
```

This call-back function is called by the stack each time a peripheral raises an interrupt. This function is called from within the interrupt context.

The DIOs may be configured to generate interrupts when the state of the inputs change. These interrupts are surfaced to the application via the System Control interrupt. These events are passed on to the DioStatus MIB via a call to the **MibDioStatus_vSysCtrl()** function for further processing.

5.2.3.7 Device_vTick() Function

```
void Device_vTick ( void );
```

This function is called each time around the main loop its main purpose is to handle the timing of the application.

The DioConfig and DioControl are updated once per second with a call to the **MibDioConfig_vSecond()** and **MibDioControl_vSecond()** MIBs.

The DioControl is updated every 10ms with a call to **MibDioStatus_vTick()**.

5.2.3.8 Device_vSleep() Function

```
void Device_vSleep ( void );
```

This function is called in End Device nodes if the main loop is allowed to exit and controls placing the node into sleep mode.

The DioConfig and DioControl MIBs are given the opportunity to save any outstanding data to the PDM before sleeping with calls to the **MibDioConfig_vSecond()** and **MibDioControl_vSecond()** functions.

5.3 Common Software

The **Common** folder of the Application Note contains source code that is used by many device types in the JenNet-IP Application Template and JenNet-IP Smart Home Application Note.

5.3.1 Config.h

Config.c contains definitions for the default network identifiers and operating parameters.

5.3.2 Node.h, Node.c

Node.c contains the core functionality for a node to join and maintain its place in the network. The functionality supports Router and End Device node types. When developing new device types the network maintenance functions in this source code module can be used as-is leaving the developer to concentrate on the functionality of the device being developed.

Most of the functionality is contained within the **MibCommon** library, there are many calls to functions in **MibCommon** that implement the required functionality.

The following sections briefly describe the features of the **Node.c** source code. For functions called during initialisation of the device they are mostly presented in the order in which they are called, though it is not a fully linear sequence.

5.3.2.1 #defines

There are a number of local #define values in **Node.c** that control the operation of the node. The most notable are described below:

```
#define MIB_NWK_SECURITY_PATCH TRUE
```

This #define when set to **TRUE** applies patches to the NwkSecurity MIB altering its operation to allow automatic switchover between gateway and standalone modes. The default value of **TRUE** is recommended.

```
#define FACTORY_RESET_MAGIC 0xFA5E13CB
```

This #define is used to verify the contents of the data used to detect the on-off factory reset power cycle sequence.

```
#define FACTORY_RESET_TICK_TIMER 32000000
```

This #define is used to time the on-off factory reset power cycle sequence.

5.3.2.2 External Data

A number of externally declared variables are used by **Node.c**.

Each MIB in the MibCommon library used by **Node.c** requires two external variables to be accessed in **Node.c**:

- Data structure, data used by the MIB is contained in a structure of the type `tsMibName` with the variable name **sMibName**, (where *Name* is the actual name of the MIB.) These data structure types are defined in the corresponding **MibName.h** include file of the application note, while the structure itself is declared in **the MibNameDec.c** source file of the application note which contains the MIB declaration.
- MIB handle, passed to JIP functions to allow access to MIBs and variables. These are of the type `thJIP_Mib` and named **hMibName** (where *Name* is the actual name of the MIB). The variable is actually declared in the **MibNameDec.c** source file of the application note.

5.3.2.3 Node_vInit() Function

```
void Node_vInit ( bool_t bWarmStart );
```

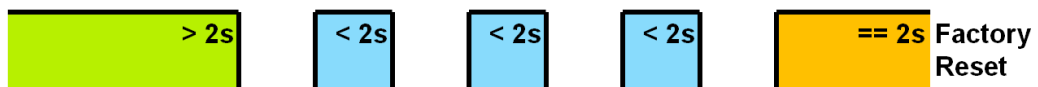
This function should be called during a Warm or Cold start to initialise the **Node.c** module. (In **DeviceTemplate.c** it is called from the **Device_vInit()** function.)

Its main purpose is to determine the type and size of the memory to store persistent data and allocate the data for use by the Persistent Data Manager (PDM), Over Network Download (OND) and factory reset detection modules and functions.

5.3.2.4 Node_bTestFactoryReset() Functions

These functions are used to check a sequence of the device being powered on and off with specific timings in order to invoke a factory reset. This may be necessary for devices without buttons or other external interfaces that can be used to generate a factory reset (such as a light bulb). While this function detects the sequence the factory reset is performed later in the boot sequence if required.

The exact sequence is to begin with the device having been powered on for a period greater than two seconds. The device should then be powered on and off so there are three consecutive periods of the device being powered for less than two seconds. The device should then be powered on one last time after two seconds the device will apply a factory reset. These timings are illustrated in the diagram below:



This strict pattern is used to reduce the risk of an accidental factory reset, especially if children might be playing with a wired light switch, with two long “guard” periods required either side of the three short periods.

In order to detect this sequence the software sets a flag upon being powered on, if the device remains on for two seconds the flag is cleared. A history of these flags is maintained in storage even when power is removed so the timing sequence can be detected. Three versions of this function are provided in the application each uses a different method to retain the data while power is off.

5.3.2.4.1 Node_bTestFactoryResetEeprom() Function (JN516x)

```
bool_t Node_bTestFactoryResetEeprom ( void );
```

The default method for JN516x devices is to store the flags in EEPROM memory. As the data is updated each time the device is powered on a whole EEPROM sector is used to store just a few bytes of data. However using EEPROM memory provides the most reliable method of retaining the flags while power is removed.

Two 32-bit values are used to validate the data stored in the EEPROM. The first is a magic number defined as `FACTORY_RESET_MAGIC`, the second is the JIP Device ID. On start up the factory reset data is read, if the validation values are incorrect the data is configured for an initial start-up and written back to the EEPROM, the remainder of the EEPROM sector is filled with junk values.

A third 32-bit value is used to track the on-off timings that trigger a factory reset, though only the least significant 8 bits are used. In a device that has been running for more than two seconds its value will be `0b11111111`. Once the flags have been read the software works from the least significant bit of the byte to the most significant bit looking for a bit set to 1. When it encounters the first bit it sets it to zero and stops looping. The new value is then written back to the EEPROM. (`0b11111110` for the first iteration, this value is also used immediately when invalid data is found in the EEPROM.)

A two second timer is started, if the timer completes the current byte value is checked and if a factory reset is not required the least significant bit is set back to 1 and written back to the EEPROM, indicating a long on cycle ready for the next iteration.

If the device is turned off while the timer is running the test byte with some bits set to zero will remain in the EEPROM and the remaining least significant set bit will be cleared on the next iteration.

If the device is powered off each time before the two second timer expires the tested byte value will have the sequence of values 0b11111110, 0b11111100, 0b11111000 as the least significant 3 bits get set to zero and stored in the EEPROM.

When the device next gets turned on and is left on when the 2 second timer expires the test byte has a value of 0b11110001 (the final bit being set after the timer expires for this last iteration). If this pattern is encountered the function return value indicates that the factory reset sequence has been detected.

If the device continues to be turned off quickly after the third iteration more bits get cleared from the test byte and the factory reset is not triggered as the byte value is never set to 0b11110001 when the timer expires.

5.3.2.4.2 Node_bTestFactoryResetFlash() Function (JN514x)

```
bool_t Node_bTestFactoryResetFlash ( void );
```

The default method for JN514x devices is to store the flags in flash memory. As the data potentially needs to be erased each time the device is powered on a whole flash sector must be used to store just a few bits of data, additionally the act of erasing the flash sector lengthens the boot sequence of the device. However using flash memory provides the most reliable method of retaining the flags while power is removed.

If a device is newly programmed or when it has been powered on for at least two seconds the flash sector will contain all 1s. The byte at the beginning of the flash sector is used to store the factory reset detection flags. In a device that has been running for more than two seconds its value will be 0b11111111.

The **Node_bTestFactoryResetFlash()** function begins by reading the byte at the beginning of the flash sector. It then works from the least significant bit of the byte to the most significant bit looking for a bit set to 1. When it encounters the first bit it sets it to zero and stops looping. The new value is then written back to the flash (0b11111110 for the first iteration).

A two second timer is started, if the timer completes the current byte value is checked and if a factory reset is not required the flash sector is erased returning the byte back to a value of 0b11111111 for the next iteration.

If the device is turned off while the timer is running the test byte with some bits set to zero will remain in the flash and the remaining least significant set bit will be cleared on the next iteration.

If the device is powered off each time before the two second timer expires the tested byte value will have the sequence of values 0b11111110, 0b11111100, 0b11111000 as the least significant 3 bits get set to zero and stored in the flash.

When the device next gets turned on and is left on when the 2 second timer expires the test byte has a value of 0b11110000 (the final bit being cleared for this last iteration). If this pattern is encountered the function return value indicates that the factory reset sequence has been detected.

If the device continues to be turned off quickly after the third iteration more bits get cleared from the test byte and the factory reset is not triggered as the byte value is never set to 0b11110000 when the timer expires.

5.3.2.4.3 Node_bTestFactoryResetWakeTimer() Function

```
bool_t Node_bTestFactoryResetWakeTimer ( void );
```

This alternative approach makes use of two registers used by the wake timers. These registers hold their contents for a short period of time after the chip is powered down. So long as the off periods of the cycle are short enough this method can be used to detect the factory reset sequence.

Unfortunately it is impossible to determine the length of time the data will be retained in these variables, as this is dependent upon power supply circuitry, potentially making this method unreliable. However it does have the advantage of speeding up the boot sequence and makes a flash or EEPROM sector available for storing other data.

A similar approach to the flash memory method is used but the bit patterns are inverted as the flash restrictions do not apply.

5.3.2.5 Node_vPdmInit() Function

```
void Node_vPdmInit ( void );
```

This function is used to initialise the PDM data and the data for each of the common MIBs implemented in the MibCommon library and used by the application.

The PDM is initialised first with a call to **PDM_vInit()**.

For JN516x devices a PDM record for the device is then read. This record simply contains the 32-bit JIP Device ID. If the Device ID read from the PDM does not match that of the running software all PDM data is deleted and the device reset. This ensures that if the software in a JN516x chip is changed the PDM data is cleared as PDM data for one device may not be compatible with another device. When the device PDM record is not present the correct data is written to the PDM. (JN514x devices do not use this mechanism as the default behaviour upon programming is to erase the whole flash memory where PDM data is stored.)

Each common MIB has its own initialisation function with each called in turn. In general the MIB initialisation functions will read their data from the PDM and initialise data and hardware as required.

5.3.2.6 Node_vReset() Function

```
void Node_vReset ( bool_t bFactoryReset );
```

This function is used to reset the device, the parameter determines whether it should be a standard reset or a factory reset.

For a standard reset the NodeStatus MIB's count of resets is incremented, all outstanding PDM data is saved. The NodeControl MIB's **Reset** variable can be used to schedule a reset.

For a factory reset the device erases the PDM data for most of the common MIBs, (the data for the NwkStatus MIB is retained in order to preserve the timer and counter variables). The NodeControl MIB's **FactoryReset** variable can be used to schedule a factory reset, the factory reset power cycle sequence may also trigger this during booting.

The device is then reset forcing it through a cold start.

5.3.2.7 Node_eJipInit() Function

```
teJIP_Status Node_eJipInit ( void );
```

This function initialises the JIP stack and registers the common MIBs.

A **tsJIP_InitData** structure is first initialised with a default set of JIP configuration values. This structure is then passed into both the **NwkConfig** and **NwkSecurity** MIBs, (with calls to **MibNwkConfig_vJipInitData()** and **MibNwkSecurity_vJipInitData()**). These functions update the initialisation structure with values stored in those MIB's PDM structures.

When running as an End Device node type the routing table size is set to zero to preserve RAM.

Next the JenNet-IP stack is initialised by calling the **eJIP_Init()** stack function, with the initialisation structure passed in as a parameter. This function will in turn call the **v6LP_ConfigureNetwork()** call-back function that must be present in the application.

Once the JenNet-IP stack is running the common MIBs used by the application are registered with the stack so they become available for access when the device joins the network.

Each common MIB has a register function, these are called in sequence to register the each MIB with the stack.

5.3.2.8 Node_v6lpConfigureNetwork() Function

```
void Node_v6lpConfigureNetwork (
    tsNetworkConfigData *psNetworkConfigData );
```

This function should be called from the stack's **v6LP_ConfigureNetwork()** call-back function which is called by the stack from the **eJIP_Init()** function during initialisation to allow the operation of the stack to be configured.

The initial section of code in **Node_v6lpConfigureNetwork()** builds a default name for the device, (if it was not retrieved from the Node MIB's PDM structure). This value is later passed into the stack's Node MIB as the value for the **Name** variable.

Next the commissioning keys used to join the network are derived from the device's MAC address, (if not retrieved from the **NwkSecurity** MIB's PDM structure). To further secure the device when joining a gateway system the gateway commissioning key could be randomly generated and provided with the device, (in an NFC tag or barcode), for out-of-band transfer to the gateway device.

The above two tasks are performed in **Node_v6lpConfigureNetwork()** as this is the earliest that the MAC address can be obtained using the **pvAppApiGetMacAddrLocation()** function. These tasks are not directly related to the configuration of the stack.

The data structure used to configure the stack is then passed into the **NwkConfig** MIB to be updated with the values stored in the **NwkConfig** MIB's PDM structure via the **MibNwkConfig_vNetworkConfigData()** function.

The **MibNwkConfigPatch_vSetUserData()** function is then used to set the Device ID and Device Type values used when the device is attempting to join a network.

The status of the device in the network in the previous power cycle is then checked and passed into the NwkSecurity MIB along with the stack's configuration data pointer via the **MibNwkSecurity_u8NetworkConfigData()** function. The NwkSecurity MIB then applies the correct security settings to join or re-join the network.

The network profile that determines the joining parameters is then configured. (This may alternatively be done using the normally disabled NwkProfile MIB).

When this function returns the JenNet-IP stack begins the process of joining or re-joining a network and control is then returned back to the application after the call to **eJIP_Init()** in **Node_eJipInit()**.

5.3.2.9 Node_v6lpDataEvent() Function

```
void Node_v6lpDataEvent ( int           iSocket,
                          te6LP_DataEvent eEvent,
                          ts6LP_SockAddr *psAddr,
                          uint8          u8AddrLen );
```

This function should be called from the stack's **v6LP_DataEvent()** call-back function which is called by the stack for data events at the 6LowPAN level. As this application is written to operate at the JIP level, reading and writing to MIB variables, any packets received from this level of the stack are simply discarded by this function.

5.3.2.10 Node_vJipStackEvent() Function

```
Bool_t Node_vJipStackEvent (
                          te6LP_StackEvent eEvent,
                          uint8            *pu8Data,
                          uint8            u8DataLen );
```

This function should be called from the stack's **vJIP_StackEvent()** call-back function which is used to inform the application of stack events relating to the status of the device in the network. The following events are handled:

E_STACK_JOINED

Indicates the device has successfully joined or re-joined a network.

The stack's Over Network Download modules are initialised with a call to the **eOND_DevInit()** or **eOND_SleepingDevInit()** function depending upon the node type. This function allocates 1Kb of memory from the heap, if an application is crashing upon joining a network it may be due to not enough memory being available for the OND buffer. Once the stack has allocated the OND buffer no further data is allocated from the heap.

MibNwkConfigPatch_vSetUserData() is called to ensure that other devices within the network and seeking to join the network can determine the device's Device ID and Device Types.

If the device has joined a standalone network for the first time and a commissioning timeout is specified the factory reset timer will be started. If the device does not complete commissioning, which ends with a cancelation of the factory reset timer, it will automatically be factory reset and have to re-join the network. This is commonly used to ensure that a newly installed device is properly programmed with group membership and other values upon joining a standalone system.

E_STACK_RESET

Indicates the device has lost contact with its network.

This event is also raised when swapping between gateway and standalone network modes. In these cases code in this function ensures that the settings for the new mode are configured correctly and the stack restarted appropriately to enter gateway or standalone mode.

E_STACK_GATEWAY_STARTED

This event is raised whenever the Border Router's announcement message is received. The Border Router transmits this message once per minute.

Devices running in standalone mode swap to trying to re-join a gateway network upon receiving this message, (this mode change is partially handled by the NwkSecurity MIB).

E_STACK_POLL

This event is raised on End Devices upon completing a request for data to its parent node.

If data is retrieved from the parent node another poll request is made using a call to **e6LP_Poll()**. This ensures that all data is retrieved from the parent node to avoid data loss due to messages timing out.

When no data is available **TRUE** is returned to the calling application allowing decisions to be made on when the End Device should sleep.

Once the event has been handled by the software in **Node.c** the stack events are also passed on to some of the other MIBs used by the application that need to be aware of the status of the device within the network or control its operation within the network. The following MIBs all have Stack Event functions; Group; NwkStatus; NwkSecurity; NwkTest; NwkProfile.

The NwkSecurity MIB includes code to determine if the device should swap between standalone and gateway mode due to the network being lost or the gateway announcing its presence.

5.3.2.11 Node_u8Analogue() Function

```
uint8 Node_u8Analogue ( uint32 u32Device,  
                        uint32 u32ItemBitmap );
```

This function should be called from the stack's **v6LP_PeripheralEvent()** call-back function (which is called by the stack each time a peripheral raises an interrupt) when an Analogue peripheral interrupt is raised. This function is called from within the interrupt context.

The AdcStatus MIB configures the ADC peripherals to take regular readings. Each time a reading is completed an interrupt will be raised.

The **MibAdcStatusPatch_u8Analogue()** function in the AdcStatus MIB is called to allow processing of the ADC reading. When the AdcStatus MIB handles a reading from the on-chip temperature sensor it will recalibrate the radio and manipulate the oscillator to compensate for changes in temperature.

This function returns ADC source of the completed reading so it may be passed to other modules used in the calling application.

5.3.2.12 Node_vTick() Function

```
void Node_vTick ( uint8   u8Tick
                  uint32  u32Second );
```

This should be called every 10ms and is usually based upon the tick timer which the stack runs at 10ms intervals. Its main purpose is to handle timing tasks in the application.

The **u8Tick** parameter provides a rolling count of 100ths of a second which loops from 0 to 99 and back again each call. This is used to time short intervals and distribute tasks across a whole second.

The **u32Second** parameter provides an incrementing counter that updates once per second while the device is running.

If a joining timeout is configured in the application the 1 second timer is used to monitor the joining time. If the timer expires without the device joining the network the stack is reset to idle to prevent the device joining.

Each Common MIB used by the application has a function that is called once per second to allow each MIB to perform its timing functions including saving their PDM data. The calls to the MIBs are distributed across the 1 second interval so they are not all called within a single tick period.

Some common MIBs need timing functions at less than one second intervals these MIBs have a tick function that is called every time the tick timer has interrupted. These MIBs are the AdcStatus and NwkTest MIBs.

5.3.2.13 Node_vSleep() Function

```
void Node_vSleep ( void );
```

This should be called before placing an End Device into sleep mode.

Each common MIB's second timing function is called to force a save of any pending PDM data.

5.3.3 AH1_EEPROM.h, AH1_EEPROM.c

Contains low level functions to directly access the EEPROM on JN516x devices. These functions are used to quickly store data when monitoring the on-off power sequence that is used in some devices to invoke a factory reset.

The use of the Persistent Data Manager (PDM) is recommended to store the majority of application data.

5.3.4 Exception.h, Exception.c

Contains exception handlers to dump exception data to flash, (and UART when debugging is enabled), for later analysis.

5.3.5 Security.h, Security.c

Implements some helper functions used to derive commissioning security keys from MAC addresses.

5.3.6 Address.h, Address.c

Implements helper functions to build group address from MAC addresses, plus data access for address MIB variables.

5.3.7 Uart.h, Uart.c

Implements minimal UART functions that efficiently output data to the UART.

For general debugging including a **Printf()** function the DBG functions included in the stack are recommended.

5.4 MibCommon Library

The MibCommon library implements a number of MIBs that can be reused in many different device types. The MIBs provide access to monitor, configure and control devices at the node and network levels plus generic ADC monitoring that recalibrates the radio if the temperature changes.

5.4.1 Changing the Library

The MibCommon library contains standard MIBs defined by NXP. If the MIB variables or their operation is changed those MIBs should be given different names and MIB IDs to avoid issues in systems and confusion for end users.

The MibCommon library is implemented in ROM on JN5142-J01 devices, changing and recompiling the library will therefore have no effect on JN5142-J01 devices.

The MibCommon library built for JN5148-J01 and JN516x chips is identical to the library in ROM on the JN5142-J01 devices. Changing the library source code for JN5148 devices is not recommended as those changes will not be applied if the same code is run in JN5142-J01 devices.

If the functionality MibCommon library needs to be altered it is recommended that this is done by replacing an entire function from the library with a new one implemented in the application. This patch function can still call into the library function if it makes sense to do so. This approach still allows the operation of the common MIBs to be altered without altering the contents of the library.

There are already quite a few examples of patched MibCommon library functions in the Application Note that implement changes required after the ROM for the JN5142-J01 was finalized. The source files that implement these functions all have **Patch** in the file name, as do the patched functions themselves. Comments in the MIB header files indicate how the patched functions should be used.

The above approach can safely be used to patch out functions that are only called externally from outside the library code. For functions that may be called from within the library additional care is needed. In some cases it is possible to override the behaviour of an internally called function by running additional code after every function call that causes an internal call, there are some examples of this in the Application Note. Where this is not practical it is possible to patch certain functions in the ROM with replacements in such a way as to allow internal calls, there are no examples of this in the Application Note.

5.4.2 Organisation

Each individual MIB is built from a number of files with a common naming scheme, example filenames are given in the form **MibName** below where **Name** should be replaced with the actual MIB's name.

When compiled the MibCommon library files are placed into the **MibCommon\Build** folder within the Application Note folder.

The source code for the MibCommon library is organized into the following folders within the **MibCommon** folder of the Application Note:

5.4.2.1 Build

The **Build** folder contains the **makefile** for the library and the library files.

5.4.2.2 Include

The **Include** folder contains a header file for each MIB implemented in the library, these header files are named **MibName.h**. These headers include the data structure definitions used by the MIB and the public function prototypes implemented by the MIB. As the MIB software in ROM is fixed the structure definitions should not be changed, any additional data required should not be added to these structures.

Each MIB also has a MIB definition header file named **MibNameDef.h**. These header files make use of JenNet-IP macro definitions to define the variables in each MIB this includes their names, data types and access flags.

MibCommon.h contains defines used throughout the MibCommon library. These are mostly MIB ID numbers, variable indices within each MIB and specific MIB variables values where limited options are available.

MibCommonDebug.h contains flags that allow different UART debugging streams to be enabled when compiling a debug version of the MibCommon library.

Table.h provides an interface to support generic access to MIB variables using the table data type.

5.4.2.3 Library

The **Library** folder contains the implementation source code for the MIBs that are compiled into the library file on JN5148 and JN5168 devices or ROM on JN5142-J01 devices. This is source code that we recommend is not changed by developers in order to prevent applications built for JN5148 devices diverging from JN5142-J01 devices.

Each MIB has a source file named **MibName.c**. These source files implement the functions required of each MIB. Many MIBs contain similar functions that carry out a common task in each MIB, (though the effects in each MIB differ). For example all MIBs contain an initialization function that is called when a device using that MIB is started, however each MIB will initialize its own data and hardware that will vary from MIB to MIB.

This folder also contains **Table.c** which contains the implementation of the generic table access functions used by some of the MIBs.

Finally **MibCommon_patch.S** contains patch points for the JN5142-J01 ROM. These patch points allow advanced developers to correctly replace functions in the ROM that are called internally by other functions in the ROM.

5.4.2.4 Source

The Source folder contains source files that are compiled into the application rather than into the library or ROM. As such these files may be changed if required.

Each MIB has a MIB declaration file named **MibNameDec.c**. These source files make use of JenNet-IP macro definitions to declare each MIB and its variables, this includes the read and write function pointers and a pointer to the data associated with each variable. These source files also instantiate each MIB's handle that is needed for various JIP functions.

There may also be patch files for MIBs that have had library functions replaced with patched functions. These are named **MibNamePatch.c**.

Finally there are some optional MIBs that were not included in the library or ROM implementations such as the NwkProfile and NwkTest MIBs. The full source code for such MIBs is included here and named **MibName.c**.

5.4.3 Node MIB

The Node MIB is mainly implemented and created in the JenNet-IP stack, however the stack does not save the contents of the Node MIB's **DescriptiveName** variable in the PDM. The Node MIB code in the MibCommon library performs the task of saving and restoring the PDM data for this MIB.

The following source files provide this functionality:

- **IncludeMibNode.h** provides the data structure types and function prototypes.
- **SourceMibNodeDec.c** declares only the data structure, (the stack contains the actual MIB declaration).
- **LibraryMibNode.c** provides the implementation.

5.4.3.1 MibNode_vInit() Function

```
void MibNode_vInit ( tSMibNode *psMibNodeInit );
```

This function initializes the Node MIB's data structure reading it from the PDM if available.

5.4.3.2 MibNode_vRegister() Function

```
void MibNode_vRegister ( void );
```

The stack actually registers the Node MIB. However this is good point to set the value of the **DescriptiveName** variable and register a call-back function so remote updates to the name can be saved to the PDM. A flag is also set to ensure that default data is written to the PDM the first time the device runs.

5.4.3.3 MibNode_vSecond() Function

```
void MibNode_vSecond ( void );
```

This function should be called once per second. The PDM data is checked and if updated the new data is written to the PDM. This limits writes of the Node MIB's data to a maximum of one write per second.

5.4.3.4 MibNode_vUpdateName() Function

```
void MibNode_vUpdateName ( char *pcName )
```

This function is registered with the stack and is called whenever the Node MIB's **DescriptiveName** variable is remotely updated. The new name is stored and a flag set to write the data to the PDM the next time **MibNode_vSecond()** is called.

5.4.4 Groups MIB

The Groups MIB is mainly implemented and created in the JenNet-IP stack, however the stack does not save the contents of the Groups MIB's **Groups** table in the PDM. The Groups MIB code in the MibCommon library performs the task of saving and restoring the PDM data for this MIB.

The following source files provide this functionality:

- **IncludeMibGroup.h** provides the data structure types and function prototypes.
- **SourceMibGroupDec.c** declares only the data structure, (the stack contains the actual MIB declaration).
- **LibraryMibGroup.c** provides the implementation.

5.4.4.1 MibGroup_vInit() Function

```
void MibGroup_vInit ( tsMibGroup *psMibGroupInit );
```

This function initializes the Groups MIB's data structure reading it from the PDM if available. The maximum number of supported groups is also passed to the stack.

5.4.4.2 MibGroup_vRegister() Function

```
void MibGroup_vRegister ( void );
```

The stack actually registers the Groups MIB. A flag is set to ensure that default data is written to the PDM the first time the device runs.

5.4.4.3 MibGroup_vSecond() Function

```
void MibGroup_vSecond ( void );
```

This function should be called once per second. The PDM data is checked and if updated the new data is written to the PDM. This limits writes of the Groups MIB's data to a maximum of one write per second.

5.4.4.4 MibGroup_vStackEvent() Function

```
void MibGroup_vStackEvent ( te6LP_StackEvent eEvent );
```

This function is called whenever a stack event is raised.

When the device joins a network for the first time after booting this function restores the group memberships that were stored by the PDM.

5.4.4.5 bJIP_GroupCallback() Function

```
bool_t bJIP_GroupCallback ( teJIP_GroupEvent eEvent,  
                             in6_addr *psAddr );
```

This stack call-back function is called whenever the device is added to or removed from a group.

This function updates the group membership list appropriately and sets the flag to ensure the updated data is written to the PDM the next time **MibGroup_vSecond()** is called.

5.4.5 NodeStatus MIB

The NodeStatus MIB provides information on the status of the node, including counts for how many times the device has booted plus watchdog and brownout errors.

The following source files provide this functionality:

- **Include\MibNodeStatus.h** provides the data structure types and function prototypes.
- **Include\MibNodeStatusDef.h** provides the MIB definition.
- **Source\MibNodeStatusDec.c** declares the data structure and MIB.
- **Library\MibNodeStatus.c** provides the implementation.

5.4.5.1 MibNodeStatus_vInit() Function

```
void MibNodeStatus_vInit (
    thJIP_Mib          hMibNodeStatusInit,
    tsMibNodeStatus *psMibNodeStatusInit );
```

This function initializes the NodeStatus MIB's data structure reading it from the PDM if available.

The status of the system at start up is checked and the MIB variables and counters are updated as appropriate.

5.4.5.2 MibNodeStatus_vRegister() Function

```
void MibNodeStatus_vRegister ( void );
```

This function registers the MIB with the stack making the variables available to be accessed by other devices.

A flag is set to ensure that default data is written to the PDM the first time the device runs.

5.4.5.3 MibNodeStatus_vSecond() Function

```
void MibNodeStatus_vSecond ( void );
```

This function should be called once per second. The PDM data is checked and if updated the new data is written to the PDM. This limits writes of the NodeStatus MIB's data to a maximum of one write per second.

5.4.5.4 MibNodeStatus_vIncrementResetCount() Function

```
void MibNodeStatus_vIncrementResetCount ( void );
```

This function should be called whenever the device is intentionally performing a software reset. It increments the reset count and sets the flag to allow data to be written to the PDM.

5.4.6 NodeControl MIB

The NodeControl MIB allows the operation of the node to be controlled, mainly providing methods to reset a node.

The following source files provide this functionality:

- **Include\MibNodeControl.h** provides the data structure types and function prototypes.
- **Include\MibNodeControlDef.h** provides the MIB definition.
- **Source\MibNodeControlDec.c** declares the data structure and MIB.
- **Library\MibNodeControl.c** provides the implementation.

5.4.6.1 MibNodeControl_vInit() Function

```
void MibNodeControl_vInit (
    thJIP_Mib          hMibNodeControlInit,
    tsMibNodeControl *psMibNodeControlInit );
```

This function initializes the NodeControl MIB's data structure reading it from the PDM if available.

The status of the system at start up is checked and the MIB variables and counters are updated as appropriate.

5.4.6.2 MibNodeControl_vRegister() Function

```
void MibNodeControl_vRegister ( void );
```

This function registers the MIB with the stack making the variables available to be accessed by other devices.

A flag is set to ensure that default data is written to the PDM the first time the device runs.

5.4.6.3 MibNodeControl_vSecond() Function

```
void MibNodeControl_vSecond ( void );
```

This function should be called once per second. The PDM data is checked and if updated the new data is written to the PDM. This limits writes of the NodeControl MIB's data to a maximum of one write per second.

This function also updates the timers used to schedule a reset or a factory reset and when the scheduled time is reached performs the appropriate reset.

5.4.7 NwkConfig MIB

The NwkConfig MIB allows the configuration parameters of the network to be altered.

The following source files provide this functionality:

- **Include\MibNwkConfig.h** provides the data structure types and function prototypes.
- **Include\MibNwkConfigDef.h** provides the MIB definition.
- **Source\MibNwkConfigDec.c** declares the data structure and MIB.
- **Source\MibNwkConfigPatch.c** provides implementations of patched functions.
- **Library\MibNwkConfig.c** provides the implementation.

This MIB is not registered with the stack by default so these variables cannot be accessed remotely. As such they operate as a set of hardcoded configuration settings.

5.4.7.1 MibNwkConfigPatch_vInit() Function

```
void MibNwkConfigPatch_vInit (
                                thJIP_Mib          hMibNwkConfigInit,
                                tSMibNwkConfig *psMibNwkConfigInit );
```

This patch function should be called instead of the **MibNwkConfig_vInit()** library function.

This function simply retains a copy of the pointer to the MIB's data structure for use in the other patch functions contained in the **MibNwkConfigPatch.c** file. It then calls the **MibNwkConfig_vInit()** library function to allow the normal initialization of the MIB.

5.4.7.2 MibNwkConfig_vInit() Function

```
void MibNwkConfig_vInit (
                                thJIP_Mib          hMibNwkConfigInit,
                                tSMibNwkConfig *psMibNwkConfigInit );
```

This function initializes the NwkConfig MIB's data structure reading it from the PDM if available.

5.4.7.3 MibNwkConfig_vJipInitData() Function

```
void MibNwkConfig_vJipInitData (
                                tsJIP_InitData *psJipInitData );
```

This function should be called before calling **eJIP_Init()** to initialize the stack. It allows the NwkConfig MIB to apply its network configuration to the JIP initialization structure.

5.4.7.4 MibNwkConfig_vNetworkConfigData() Function

```
void MibNwkConfig_vNetworkConfigData (
    tsNetworkConfigData *psNetworkConfigData );
```

This function should be called from the **v6LP_ConfigureNetwork()** function. It allows the NwkConfig MIB to apply its network configuration to the network configuration structure.

This function also calls the **MibNwkConfig_vSetNetworkId()** function, this function sets the user data used in some stack messages. The user data content has changed since the library was committed to the JN5142-J01 ROM so this must be overridden by calling the **MibNwkConfigPatch_vSetUserData()** function following the call to **MibNwkConfig_vNetworkConfigData()**.

5.4.7.5 MibNwkConfig_vRegister() Function

```
void MibNwkConfig_vRegister ( void );
```

This function registers the MIB with the stack making the variables available to be accessed by other devices.

A flag is set to ensure that default data is written to the PDM the first time the device runs.

5.4.7.6 MibNwkConfig_vSecond() Function

```
void MibNwkConfig_vSecond ( void );
```

This function should be called once per second. The PDM data is checked and if updated the new data is written to the PDM. This limits writes of the NwkConfig MIB's data to a maximum of one write per second.

5.4.7.7 MibNwkConfigPatch_bMain() Function

```
bool_t MibNwkConfigPatch_bMain ( void );
```

This patch function should be called each time around the main loop.

It completes the process of swapping from trying to join a gateway network to trying to join a standalone network when a beacon response is received indicating there is a network in standalone commissioning mode.

5.4.7.8 MibNwkConfigPatch_vSetUserData() Function

```
void MibNwkConfigPatch_vSetUserData ( void );
```

This patch function should be called following a call to **MibNwkConfig_vNetworkConfigData()** and also from **vJIP_StackEvent()** when the **E_STACK_JOINED** event is raised.

This function sets the Network ID and Device Type IDs contained in the beacon response and establish route messages transmitted by the node. Overriding or replacing the data set by the deprecated **MibNwkConfig_vSetNetworkId()** function.

This function also sets patched call-back handlers for beacon responses and network authorization.

5.4.7.9 MibNwkConfigPatch_bBeaconNotifyCallback() Function

```
bool_t MibNwkConfigPatch_bBeaconNotifyCallback (
                                tsScanElement *psBeaconInfo,
                                uint16         u16ProtocolVersion );
```

This function is called by the stack each time a beacon response message is received while trying to join a network.

The unpatched **MibNwkConfig_bBeaconNotifyCallback()** function is called first to determine if the beacon response has come from a network the device may be interested in joining.

If the beacon response is acceptable but is from a node in standalone commissioning mode but the device is trying to join a gateway network the process is begun to switch over to joining a standalone network (and completed in the next call to **MibNwkConfigPatch_bMain()**).

5.4.7.10 MibNwkConfig_bBeaconNotifyCallback() Function

```
bool_t MibNwkConfig_bBeaconNotifyCallback (
                                tsScanElement *psBeaconInfo,
                                uint16         u16ProtocolVersion );
```

This function checks whether the Network ID included in a beacon response message matches that of the network the device is trying to join. This function is called by the **MibNwkConfigPatch_bBeaconNotifyCallback()** function and so should not be registered directly with the stack.

5.4.7.11 MibNwkConfigPatch_bNwkCallback() Function

```
bool_t MibNwkConfigPatch_bNwkCallback (
                                MAC_ExtAddr_s *psAddr,
                                uint8         u8DataLength,
                                uint8         *pu8Data );
```

This function is called by the stack each time a device is attempting to establish a route while joining the network. The Network ID from the node attempting to join is compared to the Network ID of the receiving node with the node only allowed to join if the two match. This keeps unwanted nodes out of a network.

5.4.7.12 Generic Variable Set Data Functions

```
teJIP_Status MibNwkConfig_eSetUInt8 ( uint8  u8Val,
                                       void    *pvCbData );
teJIP_Status MibNwkConfig_eSetUInt16 ( uint16 u16Val,
                                       void    *pvCbData );
teJIP_Status MibNwkConfig_eSetUInt32 ( uint32 u32Val,
                                       void    *pvCbData );
```

These functions are called by the stack to set the value of some variables in the NwkConfig MIB and are specified in the MIB declaration in **MibNwkConfigDec.c**. When these functions are called a flag is set to ensure the new values are saved by the PDM. Note that the new values are not applied until the node is restarted.

These functions are used when the new value of the variable does not need to be validated.

5.4.7.13 Individual Variable Set Data Functions

```
teJIP_Status MibNwkConfig_eSetDeviceType (
                                uint8  u8val,
                                void    *pvCbData );

teJIP_Status MibNwkConfig_eSetScanChannels (
                                uint32  u32val,
                                void    *pvCbData );

teJIP_Status MibNwkConfig_eSetFrameCounterDelta (
                                uint16  u16val,
                                void    *pvCbData );
```

These functions are called by the stack to set the value of some variables in the NwkConfig MIB and are specified in the MIB declaration in **MibNwkConfigDec.c**. When these functions are called a flag is set to ensure the new values are saved by the PDM. Note that the new values are not applied until the node is restarted.

These functions are used when the new value can only take a limited set of values and needs to be validated or when the device must take some action when a variable is set.

5.4.7.14 Deprecated Functions

```
void MibNwkConfig_vSetNetworkId ( void );

void MibNwkConfig_vStackEvent (
                                te6LP_StackEvent eEvent );

bool_t MibNwkConfig_bnwkCallback (
                                MAC_ExtAddr_s *psAddr,
                                uint8          u8DataLength,
                                uint8          *pu8Data );
```

These functions should no longer be called. The description of the various patch functions above describes their alternatives.

5.4.8 NwkProfile MIB

The NwkProfile MIB contains variables that can be used to configure the network profile settings of the device. It also provides an alternative parent selection algorithm that may be used by devices trying to join the network making use of a preferred LQI threshold.

While the source code for this MIB is included in the Application Note it is not normally compiled into applications but is made available for testing and evaluation use. This MIB is not compiled into the MibCommon library but directly into the application, (where used).

The following source files provide this functionality:

- **Include\MibNwkProfile.h** provides the data structure types and function prototypes.
- **Include\MibNwkProfileDef.h** provides the MIB definition.
- **Source\MibNwkProfileDec.c** declares the data structure and MIB.
- **Source\MibNwkProfile.c** provides the implementation.

5.4.8.1 MibNwkProfile_vInit() Function

```
void MibNwkProfile_vInit (
    thJIP_Mib          hMibNwkProfileInit,
    tsMibNwkProfile *psMibNwkProfileInit );
```

This function initializes the NwkProfile MIB's data structure reading it from the PDM if available.

5.4.8.2 MibNwkProfile_vRegister() Function

```
void MibNwkProfile_vRegister ( void );
```

This function registers the NwkProfile MIB with the stack.

5.4.8.3 MibNwkProfile_vSecond() Function

```
void MibNwkProfile_vSecond ( void );
```

This function should be called once per second.

The PDM data is checked and if updated the new data is written to the PDM. This limits writes of the NwkProfile MIB's data to a maximum of one write per second.

If the data is written to the PDM the changed profile is applied to the stack by calling **MibNwkProfile_vApply()**.

5.4.8.4 MibNwkProfile_vApply() Function

```
void MibNwkProfile_vApply ( void );
```

This function applies the current profile settings to the stack.

5.4.8.5 MibNwkProfile_bScanSortCallback() Function

```
bool_t MibNwkProfile_bScanSortCallback (
    tsScanElement *pasScanResult,
    uint8          u8ScanListSize,
    uint8          *pau8ScanListOrder );
```

This function provides the alternative sorting algorithm. To use it the main application must register the call-back function with the stack by calling **vApi_RegScanSortCallback()**.

If registered this function will be called by the stack once a set of scan results have been obtained while trying to join. In this case the application implements an alternative sorting sequence.

5.4.8.6 MibNwkProfile_bScanSortCheckSwap() Function

```
bool_t MibNwkProfile_bScanSortCheckSwap (
    tsScanElement *pasScanResult,
    uint8          u8ScanListItem,
    uint8          *pau8ScanListOrder );
```

This function is called by **MibNwkProfile_bScanSortCallback()** to compare two scan results and determine if their order should be swapped.

5.4.8.7 Generic Variable Set Data Functions

```
teJIP_Status MibNwkProfile_eSetUint8 (
    uint8  u8val,
    void    *pvCbData );

teJIP_Status MibNwkProfile_eSetUint16 (
    uint16 u16val,
    void    *pvCbData );
```

These functions are called by the stack to set the value of some variables in the NwkProfile MIB and are specified in the MIB declaration in **MibNwkProfileDec.c**. When these functions are called a flag is set to ensure the new values are saved by the PDM.

These functions are used when the new value of the variable does not need to be validated.

5.4.9 NwkStatus MIB

The NwkStatus MIB allows the status of the network layer to be monitored. The source code for this MIB also ensures that the frame counter is retained and reapplied across power cycles.

The following source files provide this functionality:

- **Include\MibNwkStatus.h** provides the data structure types and function prototypes.
- **Include\MibNwkStatusDef.h** provides the MIB definition.
- **Source\MibNwkStatusDec.c** declares the data structure and MIB.
- **Library\MibNwkStatus.c** provides the implementation.

5.4.9.1 MibNwkStatus_vInit() Function

```
void MibNwkStatus_vInit (
    thJIP_Mib          hMibNwkStatusInit,
    tsMibNwkStatus *psMibNwkStatusInit,
    bool_t             bMibNwkStatusSecurity,
    uint16              u16MibNwkConfigFrameCounterDelta );
```

This function initializes the NwkStatus MIB's data structure reading it from the PDM if available.

The frame counter read from flash is increased to ensure it is greater than the value used prior to the device being powered off.

5.4.9.2 MibNwkStatus_vRegister() Function

```
void MibNwkStatus_vRegister ( void );
```

This function registers the MIB with the stack making the variables available to be accessed by other devices.

A flag is set to ensure that default data is written to the PDM the first time the device runs.

5.4.9.3 MibNodeStatus_vSecond() Function

```
void MibNodeStatus_vSecond ( void );
```

This function should be called once per second.

The frame counter in the MAC is checked and if it has advanced sufficiently from the previously saved value an update to the PDM data is forced.

The various timers used to monitor the run-time, up-time and down-time of the node in the network are updated.

The PDM data is checked and if updated the new data is written to the PDM. This limits writes of the NwkStatus MIB's data to a maximum of one write per second.

5.4.9.4 MibNwkStatus_vStackEvent() Function

```
void MibNwkStatus_vStackEvent ( te6LP_StackEvent eEvent );
```

This function is used to track if the node is a member of a network, incrementing the up counter each time a network is joined or re-joined. The status of the device in the network is also recorded and a flag set to write the data to the PDM allowing the node to be correctly restarted if power cycled.

5.4.10 NwkSecurity MIB

The NwkSecurity MIB configures and controls the network security aspects of the application. It plays a large role in the process of joining a network and swapping between gateway and standalone modes.

The following source files provide this functionality:

- **Include\MibNwkSecurity.h** provides the data structure types and function prototypes.
- **Include\MibNwkSecurityDef.h** provides the MIB definition.
- **Source\MibNwkSecurityDec.c** declares the data structure and MIB.
- **Library\MibNwkSecurityPatch.c** provides the implementation of patches to the library software.
- **Library\MibNwkSecurity.c** provides the implementation.

5.4.10.1 MibNwkSecurityPatch_vInit() Function

```
void MibNwkSecurityPatch_vInit (
    thJIP_Mib          hMibNwkConfigInit,
    tSMibNwkSecurity *pSMibNwkSecurityInit,
    bool_t             bMibNwkSecuritySecurity );
```

This patch function should be called instead of the **MibNwkSecurity_vInit()** library function.

This function simply retains a copy of the pointer to the MIB's data structure for use in the other patch functions contained in the **MibNwkSecurityPatch.c** file. It then calls the **MibNwkSecurity_vInit()** library function to allow the normal initialization of the MIB.

5.4.10.2 MibNwkSecurity_vInit() Function

```
void MibNwkSecurity_vInit (
    thJIP_Mib          hMibNwkConfigInit,
    tSMibNwkSecurity *pSMibNwkSecurityInit,
    bool_t             bMibNwkSecuritySecurity );
```

This function initializes the NwkSecurity MIB's data structure reading it from the PDM if available.

5.4.10.3 MibNwkSecurity_vJipInitData() Function

```
void MibNwkSecurity_vJipInitData (
    tsJIP_InitData *psJipInitData );
```

This function should be called before calling **eJIP_Init()** to initialize the stack. It allows the NwkSecurity MIB to note the type of node the device will be within the network so the correct security settings can be applied.

5.4.10.4 MibNwkSecurity_u8NetworkConfigData() Function

```
uint8 MibNwkSecurity_u8NetworkConfigData (
    tsNetworkConfigData *psNetworkConfigData,
    uint8                u8NwkStatusUpMode,
    uint32               u32NwkStatusFrameCounter,
    uint8                u8NwkConfigStackModeInit );
```

This function should be called from the **v6LP_ConfigureNetwork()** function. It allows the NwkSecurity MIB to apply its network security data during stack configuration.

If the device was already in a network during the previous power cycle it resumes operating in that mode. (However devices that were running in a gateway network are forced to restart in standalone mode to make them quickly available to be controlled by other devices within the same WPAN.)

For devices that have never joined a network the appropriate commissioning key is set in the stack.

5.4.10.5 MibNwkSecurity_u8ResumeGateway() Function

```
uint8 MibNwkSecurity_u8ResumeGateway (
    tsNetworkConfigData *psNetworkConfigData,
    uint8                u8NwkStatusUpMode,
    uint32               u32NwkStatusFrameCounter );
```

This function configures the security data and network data to allow the device to re-join a gateway system.

5.4.10.6 MibNwkSecurity_u8ResumeStandalone() Function

```
uint8 MibNwkSecurity_u8ResumeStandalone (
    tsNetworkConfigData *psNetworkConfigData,
    uint8                u8NwkStatusUpMode,
    uint32               u32NwkStatusFrameCounter );
```

This function configures the security data and network data to allow the device to operate in a standalone system.

5.4.10.7 MibNwkSecurity_vRegister() Function

```
void MibNwkSecurity_vRegister ( void );
```

This function registers the MIB with the stack making the variables available to be accessed by other devices.

A flag is set to ensure that default data is written to the PDM the first time the device runs.

5.4.10.8 MibNwkSecurityPatch_vSecond() Function

```
void MibNwkSecurityPatch_vSecond ( void );
```

This function should be called once per second instead of the **MibNwkSecurity_vSecond()** library function.

The **MibNwkSecurity_vSecond()** library function is first called to allow the standard handling to take place.

The network re-join timer is then checked and decremented if running. When this reaches zero the network key is deleted and the node reset. This forces the node to go through a full re-join using the commissioning key and will scan across all channels in the mask.

5.4.10.9 MibNwkSecurity_vSecond() Function

```
void MibNwkSecurity_vSecond ( void );
```

This function should not be called directly but is called from the **MibNwkSecurityPatch_vSecond()** patch function. The PDM data is checked and if updated the new data is written to the PDM. This limits writes of the NwkSecurity MIB's data to a maximum of one write per second.

5.4.10.10 MibNwkSecurityPatch_u8StackEvent() Function

```
uint8 MibNwkSecurityPatch_u8StackEvent (
                                te6LP_StackEvent    eEvent,
                                uint8                 *pu8Data,
                                uint8                 u8DataLen );
```

This patch function should be called instead of the **MibNwkSecurity_bStackEvent()** function to allow handling of stack events.

The **E_STACK_JOINED** and **E_STACK_STARTED** events are passed into the **MibNwkSecurity_bStackEvent()** function for initial handling. If the event was expected future re-join attempts are limited to the channel and PAN ID of the newly joined network, this reduces the time needed to re-join the same network.

The **E_STACK_RESET** event is raised in two different situations:

1. If the device is not currently in a network it has reached the end of a scan cycle looking for network to join and is about to restore the full scan channel mask and begin again.
2. If the device is currently in a network it indicates that contact with the network has been lost.

The **E_STACK_GATEWAY_STARTED** event is raised when the regular announcement transmitted by the border router is received. If the device is operating in standalone mode this indicates that the gateway network is available. In this case the device will attempt to re-join the gateway network.

5.4.10.11 MibNwkSecurity_bStackEvent() Function

```
bool_t MibNwkSecurity_bStackEvent (
                                         te6LP_StackEvent eEvent );
```

This function should not be called directly but is called from the replacement **MibNwkSecurityPatch_u8StackEvent()** function for **E_STACK_JOINED** and **E_STACK_STARTED** only.

When the node joins a network the network key is saved to flash and the commissioning key invalidated. If a standalone network is joined the node is taken out of commissioning mode.

5.4.10.12 MibNwkSecurity_bAddSecureAddr() Function

```
bool_t MibNwkSecurity_bAddSecureAddr (
                                         MAC_ExtAddr_s *psMacAddr );
```

This function reserves an entry in the security table for the node with the specified MAC address. This is used when the node joins a standalone network to ensure it is always able to decode messages from the node that commissioned it into the network.

5.4.10.13 MibNwkSecurity_bDelSecureAddr() Function

```
bool_t MibNwkSecurity_bDelSecureAddr (
                                         MAC_ExtAddr_s *psMacAddr );
```

This function removes an entry in the security table for the node with the specified MAC address.

5.4.10.14 MibNwkSecurity_vResetSecureAddr() Function

```
void MibNwkSecurity_vResetSecureAddr ( void );
```

This function clears all the reserved security table entries.

5.4.10.15 MibNwkSecurity_vSetSecurityKey() Function

```
void MibNwkSecurity_vSetSecurityKey ( uint8 u8Key );
```

This function is used to set the appropriate commissioning or network key for the current stack mode.

5.4.10.16 void MibNwkSecurity_vSetProfile() Function

```
void MibNwkSecurity_vSetProfile ( bool_t bStandalone );
```

This function sets the stack run profile appropriately for gateway or standalone modes.

5.4.10.17 MibNwkSecurity_eSetKey() Function

```
teJIP_Status MibNwkSecurity_eSetKey (
                                const uint8 *pu8val,
                                uint8      u8Len,
                                void        *pvCbData );
```

This function is called by the stack to set the value of the security key variables in the NwkSecurity MIB and is specified in the MIB declaration in **MibNwkSecurityDec.c**. When this function is called a flag is set to ensure the new values are saved by the PDM. Note that the new security keys are not applied until the node is restarted.

5.4.10.18 MibNwkSecurity_vGetKey() Function

```
void MibNwkSecurity_vGetKey ( thJIP_Packet  hPacket,
                              void          *pvCbData );
```

This function is called by the stack to get the value of the security key variables in the NwkSecurity MIB and is specified in the MIB declaration in **MibNwkSecurityDec.c**.

5.4.11 NwkTest MIB

The NwkTest MIB contains variables that can be used to run packet error and signal strength tests.

While the source code for this MIB is included in the Application Note it is not normally compiled into applications but is made available for testing and evaluation use. This MIB is not compiled into the MibCommon library but directly into the application, (where used).

The following source files provide this functionality:

- **Include\MibNwkTest.h** provides the data structure types and function prototypes.
- **Include\MibNwkTestDef.h** provides the MIB definition.
- **Source\MibNwkTestDec.c** declares the data structure and MIB.
- **Source\MibNwkTest.c** provides the implementation.

5.4.11.1 MibNwkTest_vInit() Function

```
void MibNwkTest_vInit ( thJIP_Mib          hMibNwkTestInit,
                       tSMibNwkTest *psMibNwkTestInit );
```

This function initializes the NwkTest MIB's data structure reading it from the PDM if available.

5.4.11.2 MibNwkTest_vRegister() Function

```
void MibNwkTest_vRegister ( void );
```

This function registers the NwkTest MIB with the stack.

5.4.11.3 MibNwkTest_vTick() Function

```
void MibNwkTest_vTick ( void );
```

This function should be called every 10ms to allow the NwkTest MIB to time the transmission of its test messages.

5.4.11.4 MibNwkTest_vStackEvent() Function

```
void MibNwkTest_vStackEvent (
                                te6LP_StackEvent    eEvent,
                                uint8                 *pu8Data,
                                uint8                 u8DataLen );
```

This function should be called to pass stack events to the NwkTest MIB. These are used to track when the device is a member of a network. Upon joining or re-joining a network the parent node's address is retained and used to transmit test messages to while the test is running.

5.4.11.5 MibNwkTest_eSetTests() Function

```
teJIP_Status MibNwkTest_eSetTests ( uint8  u8Val,  
                                     void   *pvCbData );
```

This function is called by the stack to set the value of the **Tests** variable in the NwkTest MIB and is specified in the MIB declaration in **MibNwkTestDec.c**.

The test results are reset to default values ready to be populated when the test runs.

5.4.11.6 vJIP_Remote_DataSent() Function

```
void vJIP_Remote_DataSent ( ts6LP_SockAddr *psAddr,  
                           teJIP_Status   eStatus );
```

The stack calls this call-back function to indicate the status of a transmission attempt, while the test is running a successful transmission increments the **TxOk** variable.

5.4.11.7 vJIP_Remote_GetResponse() Function

```
void vJIP_Remote_GetResponse (ts6LP_SockAddr *psAddr,  
                             uint8          u8Handle,  
                             uint8          u8MibIndex,  
                             uint8          u8VarIndex,  
                             teJIP_Status   eStatus,  
                             teJIP_VarType   eVarType,  
                             const void     *pvVal,  
                             uint32         u32ValSize );
```

This function is called by the stack to return the result of a MIB variable get request. This is the command used in the test messages so this function is called when there is a successful response. The **RxOk** variable is incremented and the LQI of the received packet measured and used to update the **RxLqiMin**, **RxLqiMax** and **RxLqiMean** MIB variables.

A successful response contains the LQI of the transmitted test packet allowing the **TxLqiMin**, **TxLqiMax** and **TxLqiMean** values to be calculated.

5.4.12 AdcStatus MIB

The AdcStatus MIB runs and monitors ADC readings as configured by the application. Any combination of ADC inputs may be configured all are read at the same rate as set by the application.

When the on-chip temperature input is used the radio is recalibrated and the oscillator pulled as required to compensate for changes in temperature.

The following source files provide this functionality:

- **Include\MibAdcStatus.h** provides the data structure types and function prototypes.
- **Include\MibAdcStatusDef.h** provides the MIB definition.
- **Source\MibAdcStatusDec.c** declares the data structure and MIB.
- **Library\MibAdcStatusPatch.c** provides the implementation of patches to the library software.
- **Library\MibAdcStatus.c** provides the implementation.

5.4.12.1 MibAdcStatusPatch_vInit() Function

```
void MibAdcStatusPatch_vInit (
    thJIP_Mib          hMibAdcStatusInit,
    tsMibAdcStatus *psMibAdcStatusInit,
    uint8              u8AdcMask,
    uint8              u8OscPin,
    uint8              u8Period );
```

This patch function should be called to initialise the MIB instead of the **MibAdcStatus_vInit()** library function.

The **u8AdcMask** parameter specifies which ADC inputs should be monitored, while the **u8Period** parameter specifies the time between each reading. All ADC inputs are read with the same period, they cannot be configured separately.

The **u8OscPin** parameter specifies which DIO should be used as an output to control the oscillator on JN5148 devices if the on-chip temperature ADC input is being monitored.

The **MibAdcStatus_vInit()** library function is called to perform the standard initialisation. Various parameters are then reinitialised and the oscillator put into the un-pulled state. A pointer to the AdcStatus MIB data structure is retained for use throughout **MibAdcStatusPatch.c**.

5.4.12.2 MibAdcStatus_vInit() Function

```
void MibAdcStatus_vInit (
    thJIP_Mib          hMibAdcStatusInit,
    tsMibAdcStatus *psMibAdcStatusInit,
    uint8              u8AdcMask,
    uint8              u8OscPin,
    uint8              u8Period );
```

This function is called from the **MibAdcStatus_vInit()** patch function.

It performs basic initialisation of the MIB's data. The analogue peripherals block is also enabled for use.

5.4.12.3 MibAdcStatus_vRegister() Function

```
void MibAdcStatus_vRegister ( void );
```

This function registers the MIB with the stack making the variables available to be accessed by other devices.

5.4.12.4 MibAdcStatus_vTick() Function

```
void MibAdcStatus_vTick ( void );
```

This function should be called every 10ms triggered by the tick timer.

A counter is maintained and the next ADC reading is started at regular intervals.

5.4.12.5 MibAdcStatusPatch_u8Analogue() Function

```
uint8 MibAdcStatus_u8Analogue ( void );
```

This patch function should be called each time an analogue interrupt is raised indicating the completion of an ADC reading.

This function stores the result making it available for access via the MIB. Results from non-JN5148 chips are scaled up to 12-bits to make them comparable to JN5148 ADC readings.

When the ADC result is from the on-chip temperature sensor it is converted to tenths of a degree Centigrade. If the temperature has changed by more than 20 degrees since the radio was last calibrated it is recalibrated. The temperature is also checked to determine if it has moved through a value that requires the oscillator to be pulled or pushed and takes action if required.

This function returns the ADC source of the reading allowing other interested source code modules to be updated with the new reading.

5.4.12.6 MibAdcStatus_vOscillatorPull() Function

```
void MibAdcStatus_vOscillatorPull ( bool_t bPull );
```

This function is used to push and pull the oscillator on JN5148 devices only.

(Direct register writes are used on JN5142-J01 and JN516x devices.)

5.4.12.7 MibAdcStatus_u16Read() Function

```
uint16 MibAdcStatus_u16Read ( uint8 u8Adc );
```

This function returns the most recent raw 12-bit reading for the specified ADC source.

5.4.12.8 MibAdcStatus_i32Convert() Function

```
int32 MibAdcStatus_i32Convert ( uint8 u8Adc,  
                                int32 i32Min,  
                                int32 i32Max );
```

This function converts and returns the most recent reading for the specified ADC. The **i32Min** and **i32Max** parameters specify the values corresponding to the minimum and maximum raw 12-bit readings. So this provides a generic conversion routine.

5.4.12.9 MibAdcStatus_i16DeciCentigrade() Function

```
int16 MibAdcStatus_i16DeciCentigrade ( uint8 u8Adc );
```

This function converts and returns the most recent reading for the specified ADC as a temperature in 10th of a degree Centigrade.

5.4.12.10 Deprecated Functions

```
uint8 MibAdcStatus_u8Analogue ( void );
```

This function should no longer be used as it has been replaced by a patch function.

5.5 MibDio Modules

The MibDio modules implement a number of MIBs that can be reused in many different device types. The MIBs provide access to configure, monitor and control the Digital IO lines of the JN516x chips.

5.5.1 Changing the Modules

The MibDio modules implement standard MIBs defined by NXP. If the MIB variables or their operation is changed those MIBs should be given different names and MIB IDs to avoid issues in systems and confusion for end users.

The MibDio modules are compiled directly into the application, (unlike the MibCommon modules which are in ROM on JN5142-J01 chips or the MibCommon library for other chips). The MibDio modules can therefore be changed as required.

5.5.2 Organisation

Each individual MIB is built from a number of files with a common naming scheme, example filenames will be given in the form **MibName** below where **Name** should be replaced with the actual MIB's name.

The source code for the MibDio modules are organized into the following folders within the **MibDio** folder of the Application Note:

5.5.2.1 Include

The **Include** folder contains a header file for each MIB implementation, these header files are named **MibName.h**. These headers include the data structure definitions used by the MIB and the public function prototypes implemented by the MIB.

Each MIB also has a MIB definition header file named **MibNameDef.h**. These header files make use of JenNet-IP macro definitions to define the variables in each MIB this includes their names, data types and access flags.

MibDio.h contains defines used throughout the MibDio modules. These are mostly MIB ID numbers, variable indices within each MIB and specific MIB variables values where limited options are available.

5.5.2.2 Source

The **Source** folder contains the source files that are compiled into the application.

Each MIB has a MIB declaration file named **MibNameDec.c**. These source files make use of JenNet-IP macro definitions to declare each MIB and its variables, this includes the read and write function pointers and a pointer to the data associated with each variable. These source files also instantiate each MIB's handle that is needed for various JIP functions.

Each MIB has a source file, these are named **MibName.c**. These source files implement the functions required of each MIB. Many MIBs contain similar functions that carry out a common task in each MIB, (though the effects in each MIB differ). For example all MIBs contain an initialization function that is called when a device using that MIB is started, however each MIB will initialize its own data and hardware which vary from MIB to MIB.

5.5.3 DioConfig MIB

The DioConfig MIB allows the operation of the Digital IO lines to be configured. The direction, pull-ups and interrupts of the IO lines can all be configured.

The following source files provide this functionality:

- **Include\MibDioConfig.h** provides the data structure types and function prototypes.
- **Include\MibDioConfigDef.h** provides the MIB definition.
- **Source\MibDioConfigDec.c** declares the data structure and MIB.
- **Source\MibDioConfig.c** provides the implementation.

5.5.3.1 MibDioConfig_vInit() Function

```
void MibDioConfig_vInit (
                        thJIP_Mib      hMibDioConfigInit,
                        tsMibDioConfig *psMibDioConfigInit );
```

This function initializes the DioConfig MIB's data structure reading it from the PDM if available.

The direction, pull-ups and interrupt settings of the Digital I/O lines are configured as specified.

5.5.3.2 MibDioConfig_vRegister() Function

```
void MibDioConfig_vRegister ( void );
```

This function registers the MIB with the stack making the variables available to be accessed by other devices.

A flag is set to ensure that default data is written to the PDM the first time the device runs.

5.5.3.3 MibDioConfig_vSecond() Function

```
void MibDioConfig_vSecond ( void );
```

This function should be called once per second. The PDM data is checked and if updated the new data is written to the PDM. This limits writes of the DioConfig MIB's data to a maximum of one write per second.

5.5.3.4 Individual Variable Set Data Functions

These functions are called by the stack to set the value of some variables in the DioConfig MIB and are specified in the MIB declaration in **MibDioConfigDec.c**. When these functions are called a flag is set to ensure the new values are saved by the PDM.

These functions are used when the new value can only take a limited set of values and needs to be validated or when the device must take some action when a variable is set.

```
teJIP_Status MibDioConfig_eSetDirection (
                                uint32  u32val,
                                void      *pvCbData );

teJIP_Status MibDioConfig_eSetPullup (
                                uint32  u32val,
                                void      *pvCbData );

teJIP_Status MibDioConfig_eSetInterruptEnabled (
                                uint32  u32val,
                                void      *pvCbData );

teJIP_Status MibDioConfig_eSetInterruptEdge (
                                uint32  u32val,
                                void      *pvCbData );
```

The above functions operate by writing directly to JN51xx registers. As such they configure all the Digital I/O pins in a single operation.

```
teJIP_Status MibDioConfig_eSetDirectionInput (
                                uint32  u32val,
                                void      *pvCbData );

teJIP_Status MibDioConfig_eSetDirectionOutput (
                                uint32  u32val,
                                void      *pvCbData );

teJIP_Status MibDioConfig_eSetPullupEnable (
                                uint32  u32val,
                                void      *pvCbData );

teJIP_Status MibDioConfig_eSetPullupDisable (
                                uint32  u32val,
                                void      *pvCbData );

teJIP_Status MibDioConfig_eSetInterruptEnable (
                                uint32  u32val,
                                void      *pvCbData );

teJIP_Status MibDioConfig_eSetInterruptDisable (
                                uint32  u32val,
                                void      *pvCbData );

teJIP_Status MibDioConfig_eSetInterruptRising (
                                uint32  u32val,
                                void      *pvCbData );

teJIP_Status MibDioConfig_eSetInterruptFalling (
                                uint32  u32val,
                                void      *pvCbData );
```

The above functions operate by calling the Application Hardware Interface (AHI) functions. As such they may be used to configure a subset of the Digital I/O pins leaving other Digital I/O pins unchanged.

5.5.4 DioStatus MIB

The DioStatus MIB provides information on the status of the Digital I/O lines, including the current input states and interrupt flags.

The following source files provide this functionality:

- **Include\MibDioStatus.h** provides the data structure types and function prototypes.
- **Include\MibDioStatusDef.h** provides the MIB definition.
- **Source\MibDioStatusDec.c** declares the data structure and MIB.
- **Source\MibDioStatus.c** provides the implementation.

5.5.4.1 MibDioStatus_vInit() Function

```
void MibDioStatus_vInit (
                        thJIP_Mib      hMibDioStatusInit,
                        tSMibDioStatus *psMibDioStatusInit );
```

This function initializes the DioStatus MIB's data structure.

The initial values of the variables are read from the JN51xx registers.

5.5.4.2 MibDioStatus_vRegister() Function

```
void MibDioStatus_vRegister ( void );
```

This function registers the MIB with the stack making the variables available to be accessed by other devices.

5.5.4.3 MibDioStatus_vTick() Function

```
void MibDioStatus_vTick ( void );
```

This function should be called every 10ms. The states of the Digital I/O input lines are read from the JN516x register. When the input states change any devices with traps on the Input variable are notified.

5.5.4.4 MibDioStatus_vSysCtrl() Function

```
void MibDioStatus_vSysCtrl ( uint32 u32Device,
                             uint32 u32ItemBitmap );
```

This function should be called when a system control interrupt is generated as input interrupts are included in this handler.

The inputs that generated the interrupt are stored in the Interrupt variable and any devices with traps are updated.

The state of the inputs are also read and used to update the Input variable. Any devices with traps on this variable are updated.

5.5.5 DioControl MIB

The DioControl MIB allows output lines of the JN51xx to be controlled.

The following source files provide this functionality:

- **Include\MibDioControl.h** provides the data structure types and function prototypes.
- **Include\MibDioControlDef.h** provides the MIB definition.
- **Source\MibDioControlDec.c** declares the data structure and MIB.
- **Source\MibDioControl.c** provides the implementation.

5.5.5.1 MibDioControl_vInit() Function

```
void MibDioControl_vInit (
                        thJIP_Mib          hMibDioControlInit,
                        tsMibDioControl *psMibDioControlInit );
```

This function initializes the DioControl MIB's data structure reading it from the PDM if available.

The output pins are restored to the state stored in the PDM data structure.

5.5.5.2 MibDioControl_vRegister() Function

```
void MibDioControl_vRegister ( void );
```

This function registers the DioControl MIB with the stack making the variables available to be accessed by other devices.

A flag is set to ensure that default data is written to the PDM the first time the device runs.

5.5.5.3 MibDioControl_vSecond() Function

```
void MibDioControl_vSecond ( void );
```

This function should be called once per second. The PDM data is checked and if updated the new data is written to the PDM. This limits writes of the DioControl MIB's data to a maximum of one write per second.

5.5.5.4 Individual Variable Set Data Functions

These functions are called by the stack to set the value of some variables in the DioControl MIB and are specified in the MIB declaration in **MibDioControlDec.c**. When these functions are called a flag is set to ensure the new values are saved by the PDM.

These functions are used when the new value can only take a limited set of values and needs to be validated or when the device must take some action when a variable is set.

```
teJIP_Status MibDioConfig_eSetOutput ( uint32  u32val,  
                                       void      *pvCbData );
```

The above function sets the state of the output pins by writing directly to the JN516x register. As such it controls all the output pins in a single operation.

```
teJIP_Status MibDioConfig_eSetOutputOn (  
                                       uint32  u32val,  
                                       void      *pvCbData );
```

```
teJIP_Status MibDioConfig_eSetOutputOff (  
                                       uint32  u32val,  
                                       void      *pvCbData );
```

The above functions turn on or off a subset of the output pins leaving other output pins unchanged. They operate by calling the Application Hardware Interface (AHI) functions.

Appendices

A Revision History

This appendix contains the revision history for the Application Note, most recent first.

A.1 04/09/2013: Public v1059

First release.

Compatibility

Product Type	Part Number	Public Version	Internal Version	Supported Chips
SDK Toolchain	JN-SW-4041	v1.1	-	JN5148-J01 JN5142-J01
JenNet-IP JN516x SDK Libraries	JN-SW-4065	-	v940	JN5168 JN5164

Important Notice

Limited warranty and liability — Information in this document is believed to be accurate and reliable. However, NXP Semiconductors does not give any representations or warranties, expressed or implied, as to the accuracy or completeness of such information and shall have no liability for the consequences of use of such information. NXP Semiconductors takes no responsibility for the content in this document if provided by an information source outside of NXP Semiconductors.

In no event shall NXP Semiconductors be liable for any indirect, incidental, punitive, special or consequential damages (including - without limitation - lost profits, lost savings, business interruption, costs related to the removal or replacement of any products or rework charges) whether or not such damages are based on tort (including negligence), warranty, breach of contract or any other legal theory.

Notwithstanding any damages that customer might incur for any reason whatsoever, NXP Semiconductors' aggregate and cumulative liability towards customer for the products described herein shall be limited in accordance with the *Terms and conditions of commercial sale* of NXP Semiconductors.

Right to make changes — NXP Semiconductors reserves the right to make changes to information published in this document, including without limitation specifications and product descriptions, at any time and without notice. This document supersedes and replaces all information supplied prior to the publication hereof.

Suitability for use — NXP Semiconductors products are not designed, authorized or warranted to be suitable for use in life support, life-critical or safety-critical systems or equipment, nor in applications where failure or malfunction of an NXP Semiconductors product can reasonably be expected to result in personal injury, death or severe property or environmental damage. NXP Semiconductors and its suppliers accept no liability for inclusion and/or use of NXP Semiconductors products in such equipment or applications and therefore such inclusion and/or use is at the customer's own risk.

Applications — Applications that are described herein for any of these products are for illustrative purposes only. NXP Semiconductors makes no representation or warranty that such applications will be suitable for the specified use without further testing or modification.

Customers are responsible for the design and operation of their applications and products using NXP Semiconductors products, and NXP Semiconductors accepts no liability for any assistance with applications or customer product design. It is customer's sole responsibility to determine whether the NXP Semiconductors product is suitable and fit for the customer's applications and products planned, as well as for the planned application and use of customer's third party customer(s). Customers should provide appropriate design and operating safeguards to minimize the risks associated with their applications and products.

NXP Semiconductors does not accept any liability related to any default, damage, costs or problem which is based on any weakness or default in the customer's applications or products, or the application or use by customer's third party customer(s). Customer is responsible for doing all necessary testing for the customer's applications and products using NXP Semiconductors products in order to avoid a default of the applications and the products or of the application or use by customer's third party customer(s). NXP does not accept any liability in this respect.

Export control — This document as well as the item(s) described herein may be subject to export control regulations. Export might require a prior authorization from competent authorities.

NXP Laboratories UK Ltd
(Formerly Jennic Ltd)
Furnival Street
Sheffield
S1 4QT
United Kingdom

Tel: +44 (0)114 281 2655
Fax: +44 (0)114 281 2951

For the contact details of your local NXP/Jennic office or distributor, refer to:

www.nxp.com/jennic