JenNet-IP Application Template Application Note

Contents

About this Manual

This manual provides information about the *JenNet-IP Application Template Application Note (JN-AN-1190)*. This Application Note provides 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:

- Section 1 "Introduction" provides an overview of the Application Note

- Section 2 "System Concepts" describes the features of a JenNet-IP system at a high level.

- Section 3 "Device Concepts" describes the features of the devices implemented in the Application Note at a high level.

- Section 4 "System Operation" describes how to operate the devices in the Application Note as an end user.

- Section 5 "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 applications to monitor and control devices in the WPAN from devices inside or outside the WPAN should refer to this chapter.

- Section 6 "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 and variables 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
CCT	Colour Controlled Temperature
DIO	Digital Input/Output
GUI	Graphical User Interface
HS	Shorthand for the hue and saturation of the HSV colour space
HSV	Hue, saturation, value colour space
IP	Internet Protocol
LAN	Local Area Network
LED	Light Emitting Diode
LQI	Link Quality Indication
MIB	Management Information Base
OND	Over Network Download
PDM	Persistent Data Manager
RGB	Red/Green/Blue
SDK	Software Developer's Kit
WAN	Wide Area Network
WPAN	Wireless Personal Area Network
XY	Shorthand for the xy components of the CIE xyY colour space.
xyY	Colour representation in the CIE xyY colour space.
XYZ	Colour representation in the CIE XYZ colour space.

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
JN516x Evaluation Kit	JN516x EK001	-	-	JN5168 JN5164
SDK Toolchain	JN-SW-4141	v1111	v1111	JN5168 JN5164
JenNet-IP JN516x SDK Libraries	JN-SW-4165	v1.2	v1107	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-3098: Beyond Studio for NXP Installation and User Guide

Provides information on installing and using the Software Developer's Kit.

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 make up the JenNet-IP WPAN 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 API functions used to program the JN516x on-chip peripherals.

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 JenNet-IP device examples based upon *JenNet-IP Application Template (JN-AN-1190)*. These examples are focused upon a Smart Lighting system.

JN-AN-1190: JenNet-IP Application Template

Provides template software to use a basis for developing Smart Devices.

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 (LAN) and even a Wide Area Network (WAN) such as the internet.

The Application Note includes software for the following Smart Devices:

- Template device 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) device 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 to create new Smart Devices from the template.

2 System Concepts

This section covers the general concepts of a JenNet-IP system.

JenNet-IP networks can operate in one of three 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 and sensors. This system provides the most flexibility and options for controlling and monitoring Smart Devices.

- Coordinator Mode replaces the gateway device with a simple Coordinator device. This effectively creates a network from only the low power wireless Smart Devices and so does not allow connections to a local IP network or the internet.

- 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 *JenNet-IP Smart Home (JN-AN-1162)*. 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 and illustrations in this section are taken from *JenNet-IP Smart Home (JN-AN-1162)* 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 *JenNet-IP Smart Home (JN-AN-1162)*:
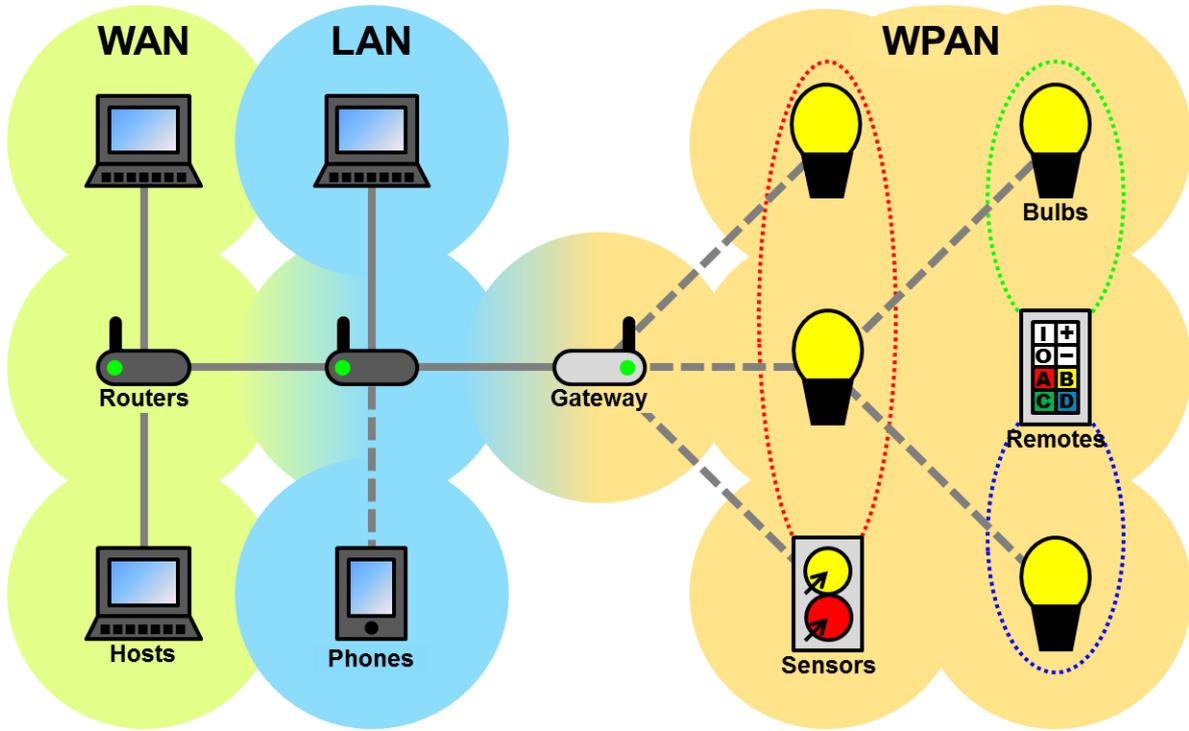
The components of the system are as follows:

Routers

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

Commands from devices in the LAN or WAN can be passed into the WPAN via the JenNet-IP gateway using cabled Ethernet connections or Wi-Fi links as shown by the solid and dotted grey lines in the LAN and WAN domains of the gateway system topology diagram.

Gateway

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.

Commands sent to individual devices in the WPAN follow the tree structure of the JenNet-IP network, (represented by the dotted grey lines in the WPAN domain of the gateway system topology diagram).

Commands broadcast to groups of devices are simply broadcast to every device in range of the original transmission, receiving devices then re-broadcast the commands ensuring that they reach every device in the network. Only the devices that are members of the group the command is addressed to will take any action (such as turning on a bulb) upon receipt of a group broadcast (though all devices re-broadcast to ensure the command reaches all devices in the network).

Smart Devices

- 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.

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

- 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, thus preserving power, only waking to read button inputs. Commands are always broadcast to a group of devices so the remote control does not need to maintain a full connection to the network. This allows the remote control to be freely moved around the area covered by the WPAN.

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 from within the wireless PAN and any connected LAN or WAN.

2.1.1 Gateway GUIs

The gateway provides a number of GUIs, served as webpages to allow administration, monitoring and control of the Smart Devices.

The gateway's landing page, accessed by entering the IP address of the gateway into a connected browser, provides links to the various GUIs:
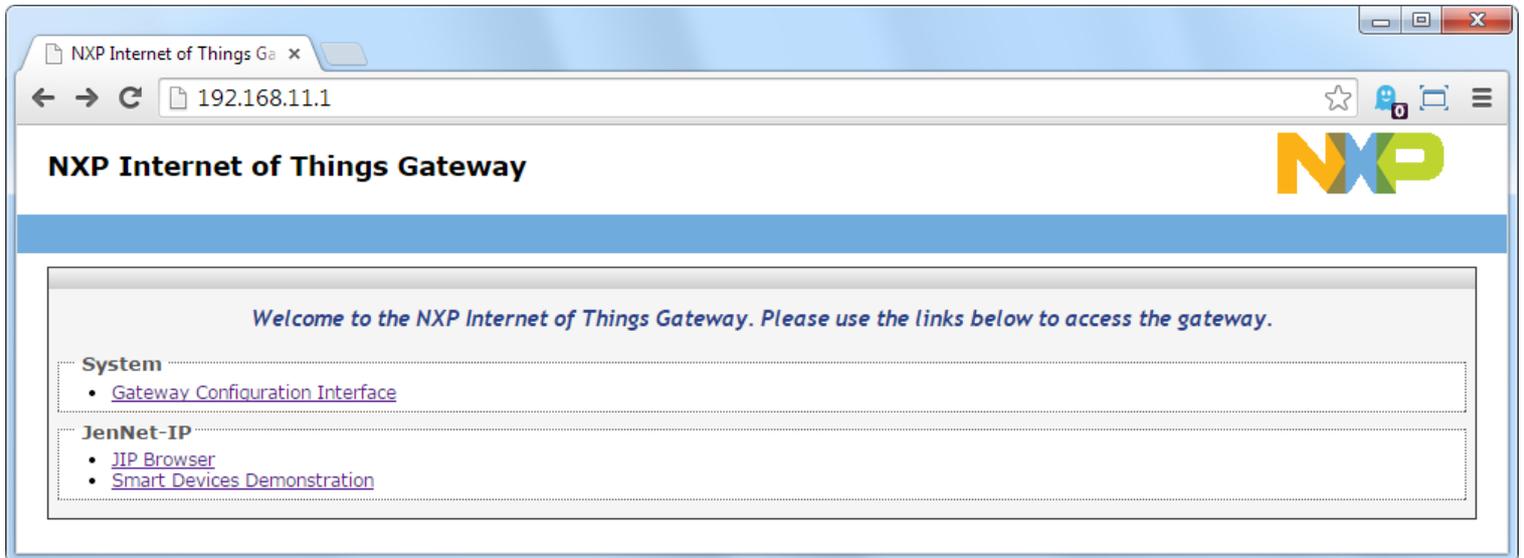
2.1.1.1 Gateway Configuration Interface

The gateway contains an Administrator GUI, based upon OpenWRT, served as a series of webpages that allow configuration of WPAN network settings.

The Administrator GUI interface also includes authorisation modules that are used to control which devices are allowed to join the WPAN. Devices attempting to join are placed into a grey list so the user is aware of their presence. Devices in the grey list can be authorised to join by the user in which case they are placed into a white list and allowed to join the network.
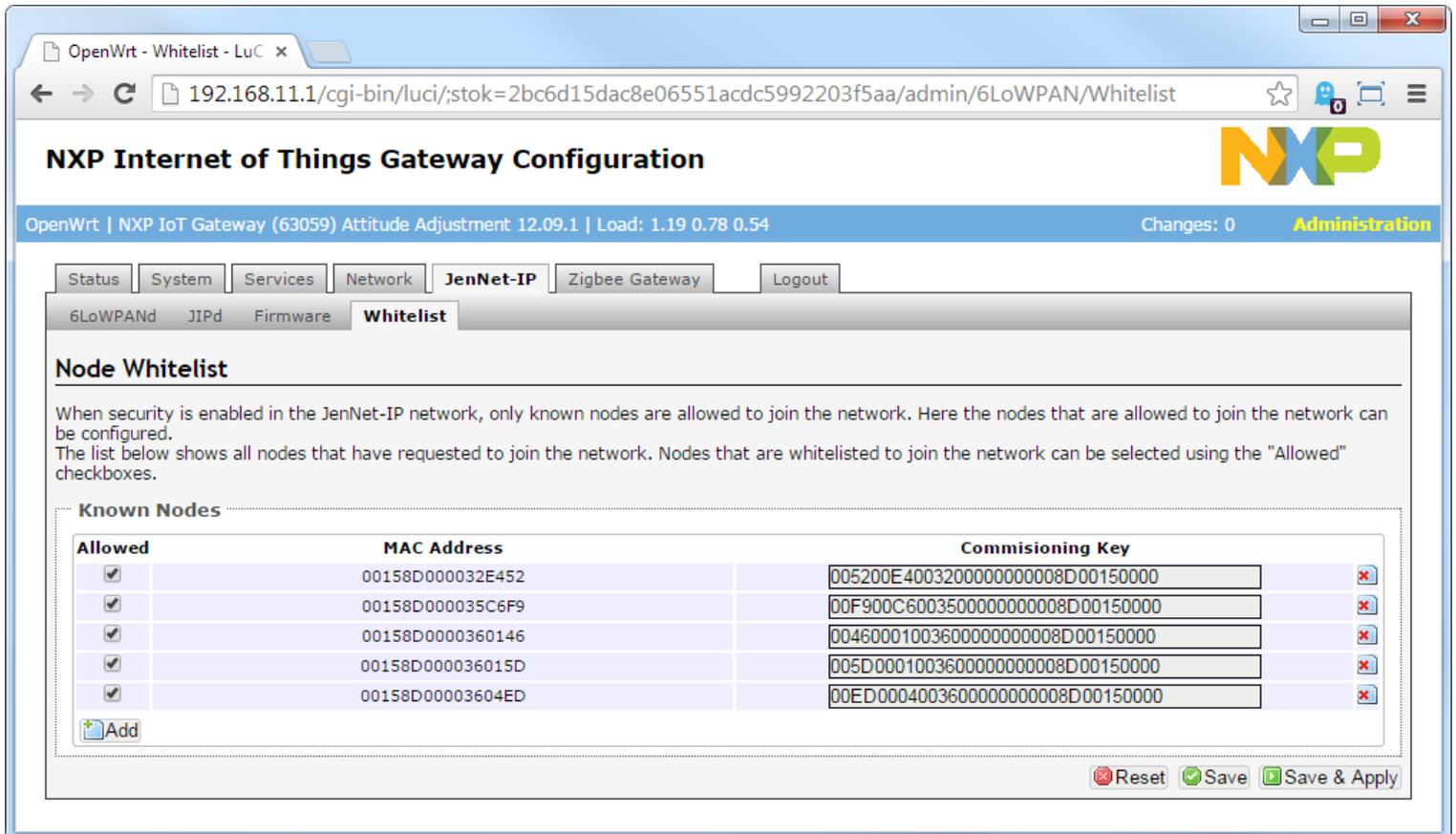
The white list includes the MAC addresses of the devices allowed into the network and a commissioning key for each device. The commissioning key in the white list must match the key programmed into the device attempting to join the network or it will not be able to join.

The devices in this Application Note use a commissioning key derived from the device's MAC address which allows the key to be pre-populated when a device is grey listed. However this is potentially insecure, a more secure solution would be to provide a random key out of band. The most convenient method for doing this would be to include an NFC tag on the device or its packaging that can be scanned into the gateway prior to installation. Other possible methods would be to print the key on the device and enter it into the gateway when white listing the device.

2.1.1.2 Gateway JIP Browser

The gateway contains a generic engineering JenNet-IP Browser interface. This allows the devices in the WPAN to be discovered. The MIBs and variables in each device can be accessed allowing them to be viewed and edited.
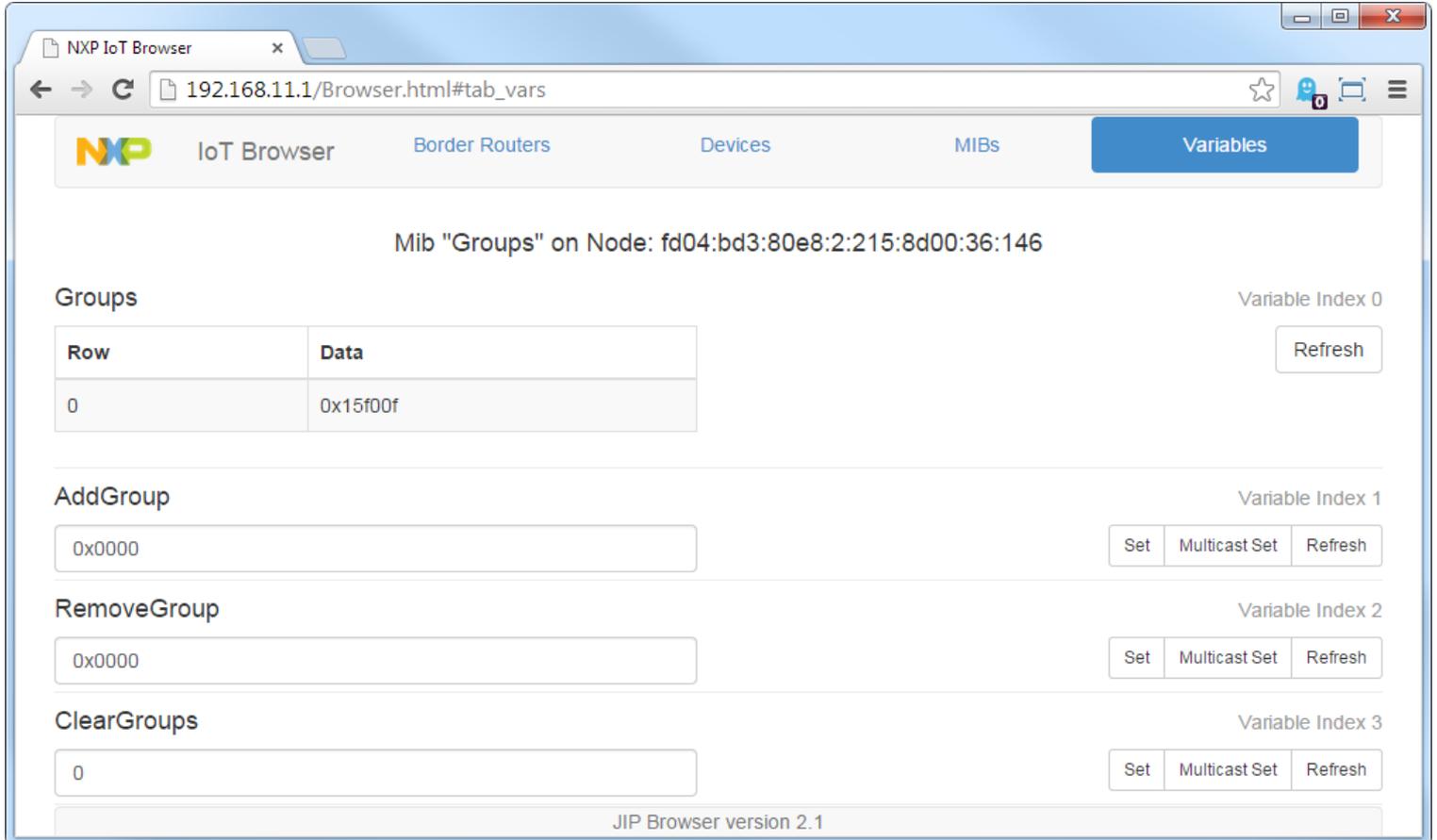
This interface is a convenient way to explore how devices can be monitored and controlled before writing applications or creating new devices that need to interact with other devices in the network, as every variable in every device can be easily accessed.

2.1.1.3 Gateway Smart Devices Demonstration

The Smart Devices GUI serves webpages specifically designed to control bulb devices. Bulbs can easily be turned on or off and the brightness level and colour of the bulbs can be set.
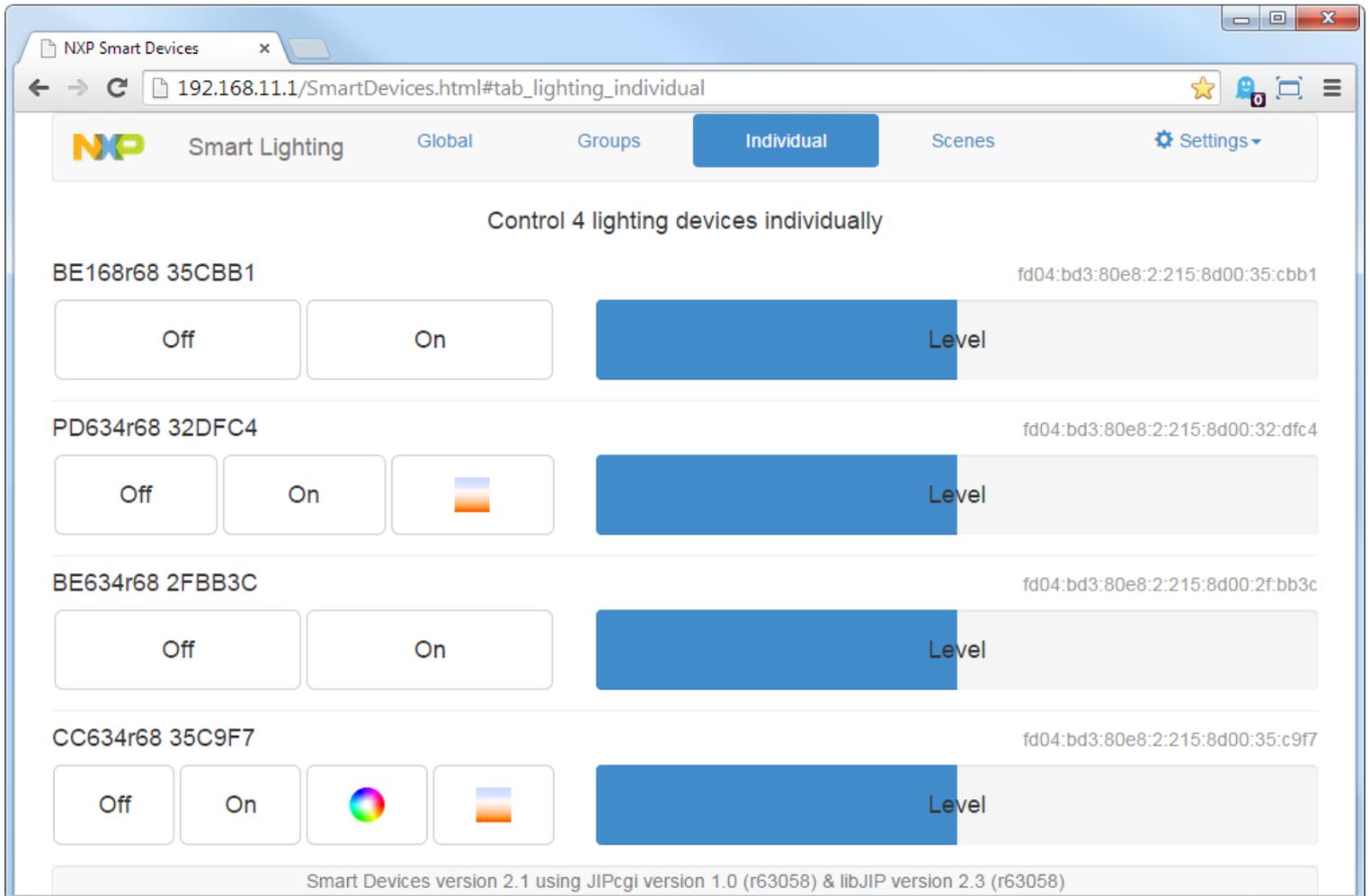
Groups of bulbs may be controlled together to set them all with the same settings. Finally bulbs can be placed into scenes which allow them to be configured with different settings but to have those settings activated at the same time by broadcasting a single command through the network, (and so avoid having to send a separate command to each bulb).

> The Smart Devices GUI does not provide an interface to place bulbs into groups or configure scenes, the JenNet-IP Browser GUI must be used for this purpose.
>
> The Smart Devices GUI only provides control of bulbs, other device types must be controlled using the JenNet-IP Browser GUI.

2.1.2 Gateway Hardware

The Internet Router/JenNet-IP Gateway hardware is formed from three components:

- Internet Router: This provides standard IP routing services, in most configurations this is an off-the-shelf device running the stock manufacturer's firmware.

- Border Router Host: This runs a version of the Linux based OpenWRT system customised by NXP to allow the management of JenNet-IP networks. Connected to the internet router via Ethernet or Wi-Fi. The software for the border router host is described in *JenNet-IP Border Router (JN-AN-1110)*.

- Border Router Node: This runs on a JN5168 device and provides the low power radio services to the border router host. It is connected to the border router host via a serial link. The software for the border router node is described in *JenNet-IP Border Router (JN-AN-1110)*.

These three components can be combined in a number of different configurations:

2.1.2.1 Internet Router with Custom Firmware

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

- The internet router is a standard off-the-shelf router running NXP's customised version of the Linux based OpenWRT. This allows both standard internet routing software and the border router host software to run as a single package in one device, with both pieces of software running on the same processor. These components run on the Linksys WRT160NL in the *JN516x Evaluation Kit (EK001)*.

- The border router node runs on a second device with a serial connection to the border router host. This component runs on a JN516x USB dongle from the *JN516x Evaluation Kit (EK001)*.

The configuration isn't really suitable for end users as it requires replacing the stock firmware in a compatible commercial internet router which is often a difficult process. However it is suitable for development use as it allows the use of an easily obtainable consumer internet router in the *JN516x Evaluation Kit (EK001)*.

2.1.2.2 Combined Border Router

This is the configuration implemented in the Reference Design *JN5168/LPC3240 Gateway (JN-RD-6040)*:

- The internet router is a standard off-the-shelf router running the manufacturer's stock firmware.

- The border router host and node are combined in a single device with the host OpenWRT firmware running on an LPC3240 microcontroller and the node firmware running on a JN5168. The border router device is connected to the internet router via Ethernet (or Wi-Fi if using suitable hardware).

This configuration is most suitable for end users as it allows existing IP systems to be easily extended to include JenNet-IP devices by simply connecting the border router device to the network.

2.2 Coordinator System Topology

The following diagram shows the topology of a coordinator system built from the lighting devices in *JenNet-IP Smart Home (JN-AN-1162)*:
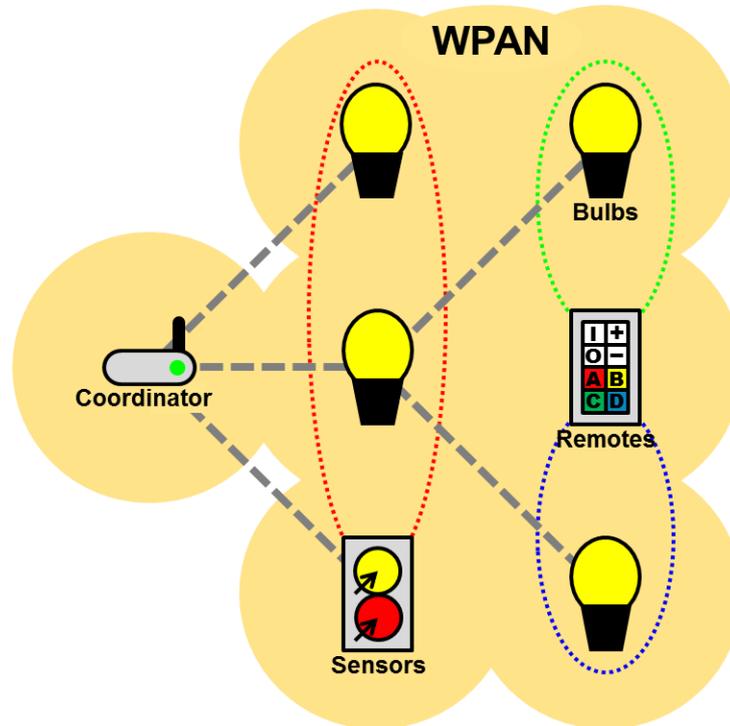
This topology replaces the gateway with a Coordinator device that does not allow connection to an existing IP network. The Coordinator device independently creates the network for the other devices to join and accepts any device attempting to join its network.

In a Coordinator 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 from within the WPAN only.

The Coordinator firmware is implemented in *JenNet-IP Application Template (JN-AN-1190)* using the Coordinator build of the template application. This firmware could be extended to allow additional control over which nodes are allowed to join the network and also implement control and/or monitoring of the other devices in the network.

2.3 Standalone System Topology

The following diagram shows the topology of a standalone system built from the lighting devices in *JenNet-IP Smart Home (JN-AN-1162)*:
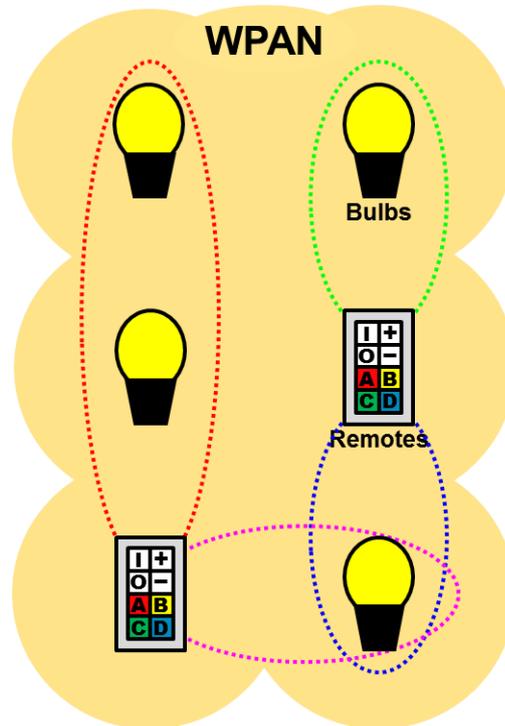
This topology does not include a Coordinator node to form the network. Instead a remote control chooses a security key for the network and can be placed into a commissioning mode using a sequence of keys. While in commissioning mode other Smart Devices in range can communicate with the remote control to retrieve the security key and 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.

The remote control firmware is implemented in *JenNet-IP Smart Home (JN-AN-1162)* using the remote control application.

2.4 Gateway/Coordinator Failure

When devices in a gateway or coordinator system lose contact with the network they will continue to receive broadcast commands and so operate in a similar way to devices in a standalone system. 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 as long as broadcast messages are used. This situation may occur if a gateway or coordinator device is powered off.

2.5 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 types while other MIBs are specific to certain device types.

The template software included in *JenNet-IP Application Template (JN-AN-1190)* 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.

Section 5 "MIB Variable Reference" covers each MIB and variable implemented in this Application Note in detail for comprehensive information.

2.6 Custom Protocols

The use of MIB and variables provides a standardised device-centric way to monitor and control devices. Development of a new application is focussed on the functionality to be provided by each device rather than on developing wireless protocols.

In some scenarios it may be necessary to develop custom wireless protocols. This can be achieved in JenNet-IP by opening additional sockets in devices, then sending and receiving messages at the 6LoWPAN layer of the stack.

When using 6LoWPAN layer messaging in this way, it is also possible to make use of JenNet-IP MIBs and variables for additional flexibility.

The *JenNet-IP Application Template (JN-AN-1190)* includes the DeviceProtocol application (which can be built as a Coordinator, Router or End Device) that illustrates how to do this.

> (i) **Note:** The *JenNet-IP Border Router (JN-AN-1110)* only opens the socket used for MIB and variable based communications. If the border router needs to send or receive 6LoWPAN level messages it will be necessary to adapt the border router firmware.

© NXP Laboratories UK 2015

2.7 Identifiers

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

2.7.1 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 then a new Device ID should be allocated.*
>
> **!** *When changing the MIBs in a device during development, it may be necessary to power down the changed device and reset the gateway to clear any information cached in the gateway for the device.*

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

The Device ID is divided into three components, described below:

2.7.1.1 Sleeping Device Flag (1 bit)

The most significant bit is used to indicate whether 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 to receive further messages and thus improve the responsiveness of the End Device when many messages need to be sent. This bit can be used to identify such devices.

2.7.1.2 Manufacturer ID (15-bits)

The next most significant 15 bits represent 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 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.

2.7.1.3 Product ID (16 bits)

The least significant 16 bits represent 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), they may allocate Product IDs as they see fit.

2.7.2 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 decide which action to take 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 of a type 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, as described below:

2.7.2.1 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.

2.7.2.2 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.

2.7.3 MIB IDs (32 bits)

Each MIB has a 32-bit MIB ID which provides a convenient way to access MIBs irrespective 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 then a new MIB ID should be allocated.*
>
> **!** *When changing the variables in a device during development, it may be necessary to power down the changed device and reset the gateway to clear any information cached in the gateway for the changed MIB.*

There are two types of MIB IDs, as described below:

2.7.3.1 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 a standard MIB must include the variables specified for that MIB by NXP. If the variables in such a MIB are adapted by the customer then the standard MIB ID should be replaced with a Manufacturer MIB ID to maintain standardisation.

2.7.3.2 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.

2.8 Message Transmission

There are two different ways to transmit messages in a JenNet-IP network, as described below:

2.8.1 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 control 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 control is able to use unicasts to commission devices.

2.8.2 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 will act upon the received message. The groups to which each node belongs can be configured using the stack's Groups MIB.

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

These messages are typically used by the remote control to control devices. Multicasts may also be issued from or via the gateway or sensors to control groups of nodes.

3 Device Concepts

This section covers the concepts of the devices implemented in this Application Note:

3.1 Template

The template devices provide the minimal functionality to get a node running in a JenNet-IP network. As such they are intended to be copied and built upon in order to create devices that provide specific functionality.

The template software can be built as a Coordinator (for use where a gateway is undesirable), Router or End Device.

3.2 Digital I/O

The digital I/O device builds on the template software to provide an example of how to extend the template. MIBs implemented in this device allow full control of the digital I/O pins of the JN516x chip.

The digital I/O software can be built as a Router or End Device.

4 System Operation

This section describes the operation of the devices implemented in the Application Note.

The software is written to run on the hardware included in the *JN516x Evaluation Kit (EK001)*. The hardware provided in the evaluation kit is separately described in *JN516x EK001 Evaluation Kit User Guide (JN-UG-3093)*.

The devices in this Application Note support three 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

- Coordinator system in which the nodes of a WPAN can be controlled only from a wireless device within the WPAN (there is no external IP connection) using unicasts and/or broadcasts.

- Standalone system in which the nodes of a WPAN can be controlled only from a wireless device within the WPAN (there is no external IP connection) using broadcasts.

The devices in the *JenNet-IP Application Template (JN-AN-1190)* and *JenNet-IP Smart Home (JN-AN-1162)* Application Notes are capable of operating in all three types of systems. However, due to the nature of the included applications there may be limitations in the functionality when used in certain system types.

Gateway System

The devices in both these Application Notes are fully functional with access to all features when used in a gateway system using the lighting GUIs and the generic JenNet-IP browser implemented in the border router host software.

Coordinator System

The Coordinator device is implemented by the template device in *JenNet-IP Application Template (JN-AN-1190)*. This Coordinator device will accept all the devices types in the Application Notes into its network except for the low energy switch devices in *JenNet-IP Smart Home (JN-AN-1162)*.

However as there is no method to provide general access to MIB variables this template is provided as a simple example of how to create such a system.

Standalone System

The remote control device is implemented in *JenNet-IP Smart Home (JN-AN-1162)*. The remote control is only able to accept bulb devices into its network. *JenNet-IP Smart Home (JN-AN-1162)* describes how to set up a standalone system using only these devices.

However, as there is no method to provide general access to MIB variables the functionality is limited to using the remote control to control bulb devices.

4.1 Gateway System Operation

This chapter describes how to use the contents of the *JN516x Evaluation Kit (EK001)* to set up and run the *JenNet-IP Application Template (JN-AN-1190)* . 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 it 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.

4.1.1 Gateway System Operation Overview

In the *JenNet-IP Application Template (JN-AN-1190)* 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 programmed as a WPAN Router or End Device. 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 the border router node and WPAN Coordinator. The dongle connects to the Linksys router (via the USB extension cable). Together they provide the gateway 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 Coordinator 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 is programmed with an NXP firmware upgrade, based upon OpenWRT, which allows the router to operate as the border router host and a standard IP router. It is connected to the above USB dongle (via the USB extension cable). Together they provide the gateway 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.

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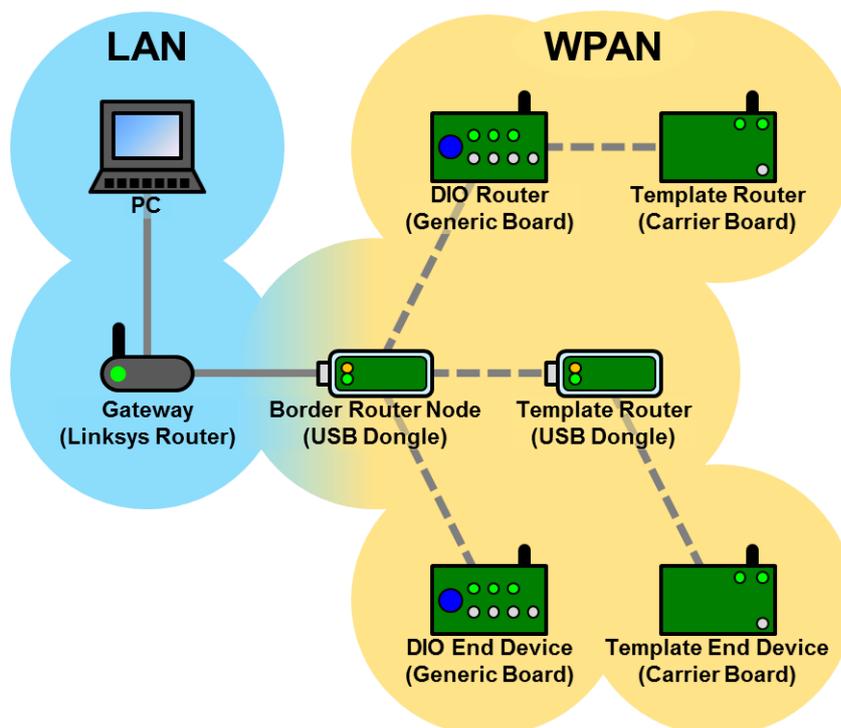
The WPAN will have a tree topology but its precise topology cannot be pre-determined since the network is formed dynamically. One or more of the Routers may be leaf-nodes of the tree, in which case their routing capability will not be used.

4.1.1.1 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/Browser.html

The JenNet-IP Browser can be accessed via the gateway's landing page by simply entering the IP address of the gateway into a web browser on the PC:

http://192.168.11.1/

The other interfaces provided by the gateway, the Smart Devices Demonstration and the Gateway Configuration interfaces, are also accessible from the gateway's landing page. These interfaces are described later in in this document.

4.1.2 Setting Up the Gateway System

This section describes the general procedures for setting up the gateway system using the evaluation kit components, instructions for specific device types are included in later sections of the Application Note.

4.1.2.1 Programming the Device Firmware

To run the software in this Application Note, the appropriate firmware must be programmed into the evaluation kit hardware.

> (i) 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 the IDE and compiling are included in *Beyond Studio for NXP Installation and User Guide (JN-UG-3098).*
>
> (i) Instructions on how to connect the *e*valuation *k*it boards to a PC and program them with firmware are included in *Beyond Studio for NXP Installation and User Guide (JN-UG-3098).*
>
> (i) If the board has previously been used, it will retain settings (e.g. PAN ID) from the previous network to which it belonged, which may prevent it joining the JenNet-IP network. These settings may be cleared during programming by erasing the EEPROM data in the device.

Pre-built binary files are included in the **Binary** folder of the Application Note for programming into the evaluation kit boards. The binaries to be used are specified in the sections covering each device type later in this document.

4.1.2.2 Setting Up the Border Router

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 border router part of the system, follow the instructions below.

Step 1   **Program the border router node firmware into a USB dongle (once only)**

If the USB *d*ongle is not already programmed with the latest *b*order *r*outer *n*ode firmware it should be updated prior to running the *a*pplication.

(i) The USB *d*ongles provided in the *e*valuation *k*it are pre-loaded with an older version of the *b*order *r*outer *n*ode firmware and should be updated to the latest version before using the Application Note. *The Application Note JenNet-IP Border Router (JN-AN-1110)* contains the *b*order *r*outer *n*ode firmware.

(i) Instructions on how to connect the USB *d*ongles to a PC and program them with firmware are included in *Beyond Studio for NXP Installation and User Guide (JN-UG-3098)* .

Step 2   **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 3   **Connect the USB dongle to the Linksys router**

Connect the USB dongle (which is programmed as a border router node and as a WPAN Coordinator) 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 4   **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 Coordinator of the WPAN, this device will create a network, for the moment consisting of just the Coordinator - the rest of the network will be formed later.

Step 5   **Program the Border Router Host firmware into the Linksys router (once only)**

If the Linksys Router is not already programmed with the latest *b*order *r*outer *h*ost firmware it should be updated before continuing to run the *a*pplication.

> (i)   The Linksys router provided in the *e*valuation *k*it is pre-loaded with an older version of the *b*order *r*outer *h*ost firmware and should be updated to the latest version before using the Application Note.
>
> (i)   The *Application Note JenNet-IP Border Router (JN-AN-1110)* contains the *b*order *r*outer *h*ost firmware and instructions to update the Linksys router.

Step 6   **Enable JenNet-IP in the Linksys router from the PC (once only)**

The latest border router host firmware is able to work with JenNet-IP or ZigBee PRO low power wireless networks. The Linksys router firmware must be enabled for use with a JenNet-IP system before continuing to use the application.

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/ then click the **Gateway Configuration Interface** link

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

d) On the next web page, select the **ZigBee Gateway** tab, then select the **ZigBee – JIP** sub-tab.

The **ZigBee – JIP** sub-tab is illustrated in the screenshot below:
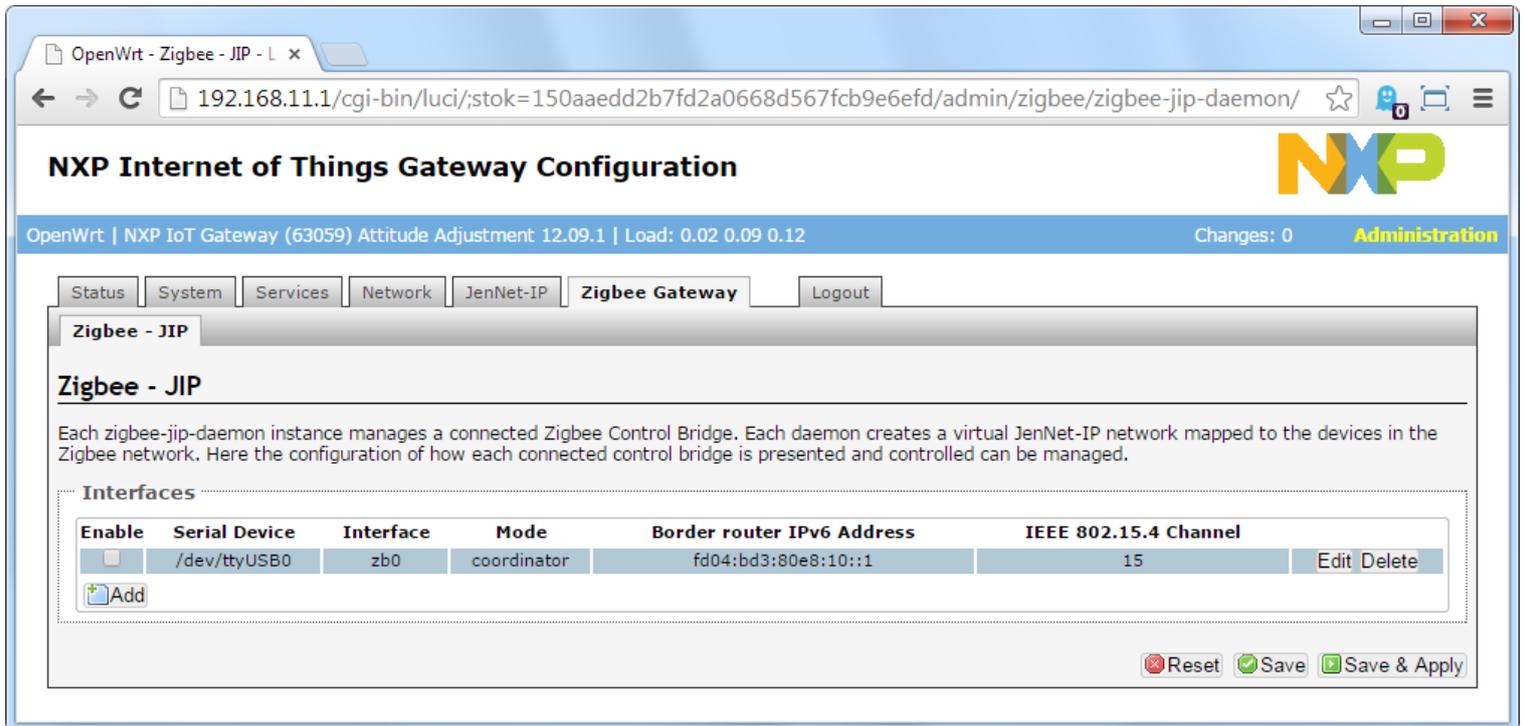
c) In the **ZigBee - JIP** sub-tab, make sure that the **Enable** checkbox is *not* ticked.

d) If the checkbox was cleared, click the **Save & Apply** button to save the changes.

e) Next select the **JenNet-IP** tab, then select the **6LoWPANd** sub-tab.

The **6LoWPANd** sub-tab is illustrated in the screenshot below:
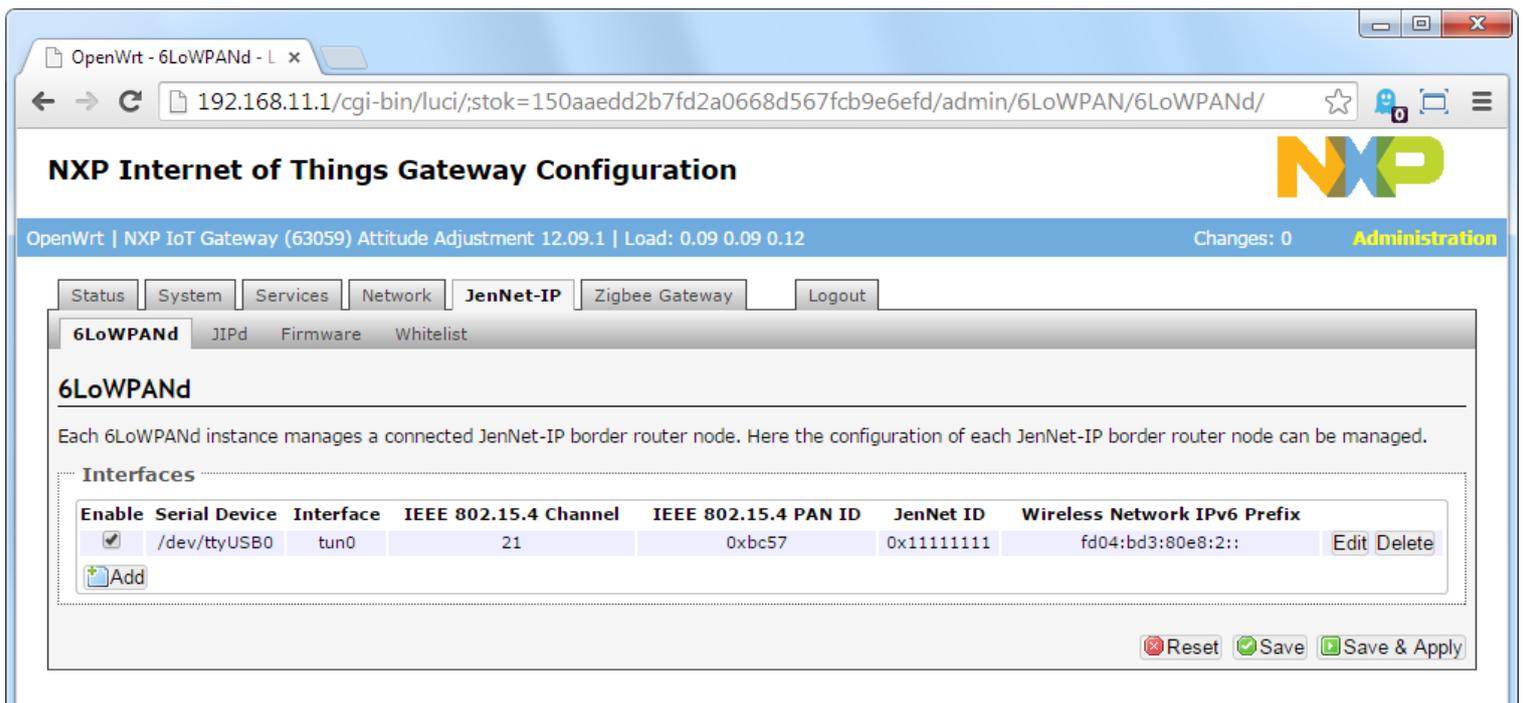
f) In the **6LoWPANd** sub-tab, make sure that the **Enable** checkbox *is* ticked.

g) If the checkbox was set click the **Save & Apply** button to save the changes.

h) Once the border router has correctly created a network the **IEEE 802.15.4 Channel** value will be non-zero and the **IEEE 802.15.4 PAN ID** will display a value different than 0xffff. The page may need to be refreshed to update this information.

Step 7   **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.

> **(i)** **Note:** The low energy switch binary included in *JenNet-IP Smart Home (JN-AN-1162)* operates only on channel 21. If you intend to add the low energy switch to your system you should channel to channel 21 at this stage.

a) Use a web browser on the PC to access the **6LoWPANd** sub-tab in the JenNet-IP Border Router Configuration interface as described in **Step 6** if it is not already open.

b) The **6LoWPANd** sub-tab is illustrated in the screenshot above.

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

Field	Description
Enable	Checkbox used to enable/disable the 6LoWPANd interface
Serial Device	Indicates serial port to which Border-Router node (dongle) is connected on the Linksys router
Interface	Indicates the network interface that hosts the JenNet-IP network.
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

c) In the **6LoWPANd** sub-tab, click the **Edit** button on the right-hand side.

d) 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:

e) 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 you intend to add the low energy switch from *JenNet-IP Smart Home (JN-AN-1162)* to the system, change the **Channel** field to 21

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

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

4.1.2.3 Adding Devices to the WPAN

This section describes the general process used to add devices to the WPAN, instructions for specific devices are included in later sections.

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

- Border router part of the system (set up as described in Section 4.1.2.2 "Setting Up the Border Router")

- Evaluation kit boards fitted with JN516x modules, optional expansion boards, antennae and batteries. These boards must be programmed with the required firmware

- (Optional) second USB dongle programmed with the required firmware

> (i) *JN516x EK001 Evaluation Kit User Guide (JN-UG-3093)* contains instructions for connecting the evaluation kit components together.

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.

General instructions applicable to all devices are below, (further information for specific device types, where applicable, are included in the following sections):

Step 1   **Start the node**

Perform the following for just one node:

On power-up, the node will attempt to join the WPAN (for which the USB dongle is the Coordinator). 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)*.

> (i) **Note:** If the board has previously been used, it will retain settings (e.g. PAN ID) from the previous network to which it belonged. This information can be cleared during programming by erasing the EEPROM data held in the device. To clear this information at run-time 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.

Step 2 **Access the Gateway Configuration interface from the PC**

If not already done (from the border router set-up), access the JenNet-IP Gateway Configuration interface from the PC as follows:

e) Launch a web browser on the PC.

f) Access the JenNet-IP landing page on the Linksys router by entering the following IP address into the browser:

http://192.168.11.1/

g) Select the **Gateway Configuration Interface** link

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

Step 3 **Display the 'whitelist' of WPAN nodes in the interface on the PC**

i) 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.
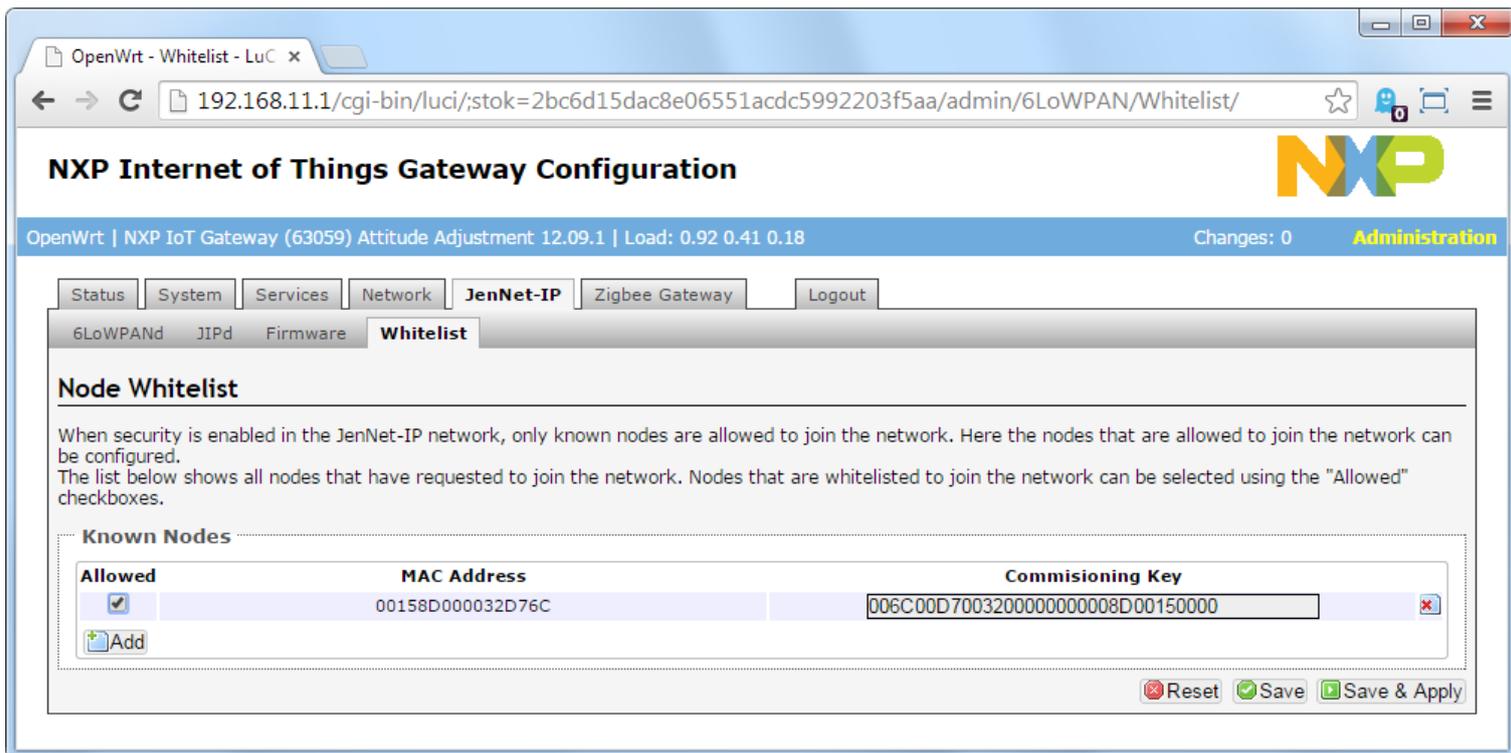
h) Put the evaluation kit board into the whitelist by ticking its checkbox on the left-hand side and click the **Save & Apply** button. The unit should now be able to join the network.

> (i) The easiest way to check whether a device has joined the network is to access the JIP Browser interface in the gateway as described in the next section.

Step 4 **Add additional nodes to the whitelist in the JenNet-IP Gateway Configuration interface**

Additional nodes can be added to the whitelist by repeating the above steps.

© NXP Laboratories UK 2015 JN-AN-1190 v2004

4.1.3 Operating the Template Devices

The template devices in the WPAN can be controlled from the PC via IP. Due to its nature as a template, this application 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.

4.1.3.1 Setting Up the Template Devices

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

- LAN part of the system (set up as described in Section 4.1.2.2 "Setting Up the Border Router".

- *Carrier Boards (DR1174)* fitted with JN516x modules, antennae and batteries programmed with the required firmware.

To set up the bulb part of the system follow the instructions below:

Step 1   **Setup template hardware**

The template device firmware can be used on any of the evaluation kit boards, the second USB dongle and the *Carrier Boards (DR1174)* are sufficient to run this software.
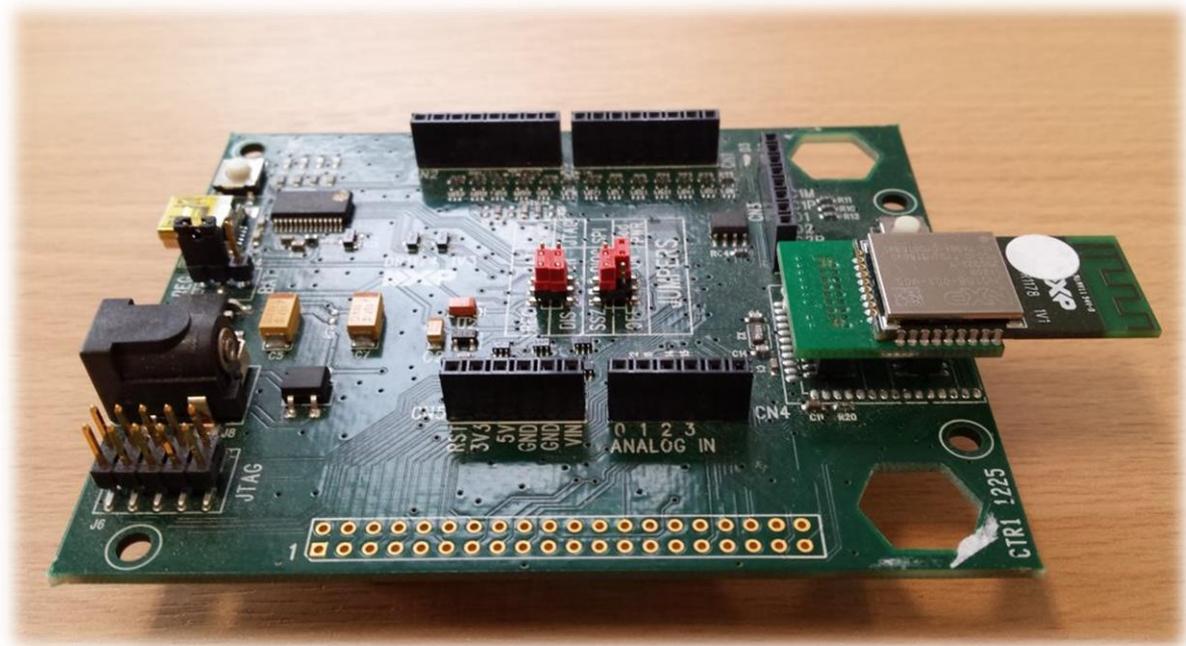
Step 2   **Program the template software**

The following pre-built binaries are provided in the Application Note for use on the evaluation kit boards.

0x11111111s_DeviceTemplate_DR1174_EndDevice_JN5168_v*0000*.bin
0x11111111s_DeviceTemplate_DR1174_Router_JN5168_v*0000*.bin

When adding these devices to a gateway system only the Router or End Device should be used (the Coordinator will create its own separate network).

> **!** *It is recommended to erase the contents of the device's EEPROM when programming the device otherwise it may retain settings for an old network and be prevented from joining the new network created in this section.*

Step 3   **Add the template devices to the network**

Follow the general procedure for powering on the boards and white listing the devices described in Section 4.1.2.3 "Adding Devices to the WPAN".

On power-up, the node will attempt to join the WPAN (for which the USB dongle is the Coordinator).

Step 4   **Template device feedback**

The template device does not provide any feedback to indicate it has joined the network. The following section describes how to access the template device from the PC which can also be used to verify it has joined.

4.1.3.2 Template Control from PC

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.

Step 1   **Access the JenNet-IP Browser**

The JenNet-IP browser is accessed by selecting the JIP Browser link on the gateway landing page at:

<div align="center">http://192.168.11.1/</div>

Step 2   **Border router node selection**

The JenNet-IP Browser first displays a list of border router nodes connected to the border router host as shown in the following screenshot:
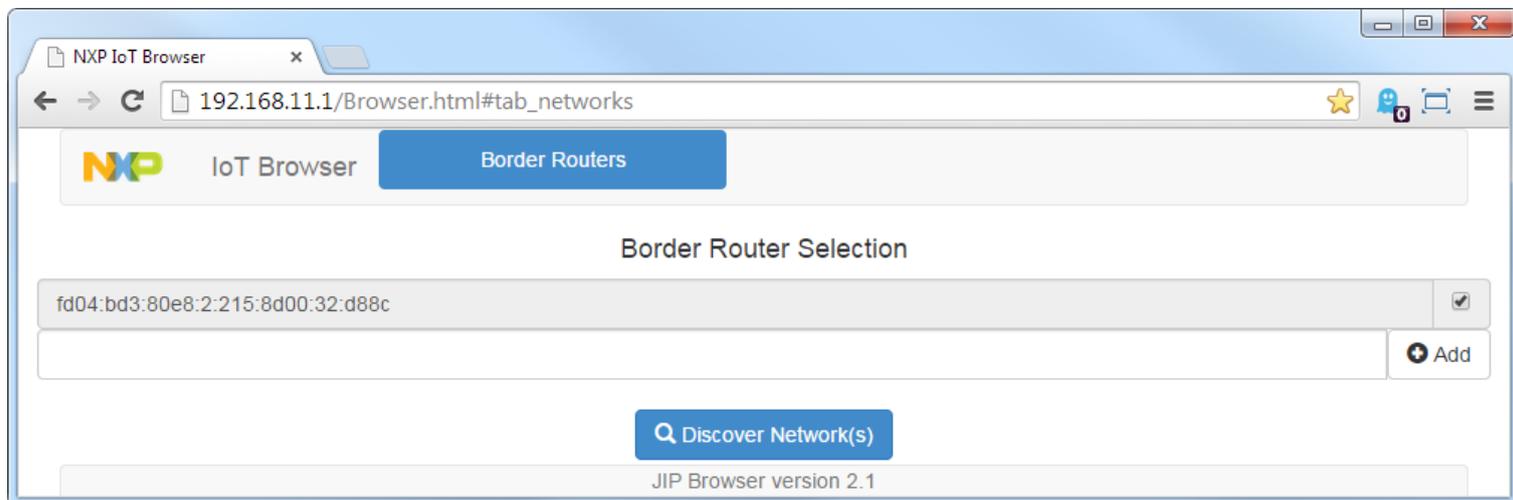
It is possible to connect more than one border router node to the host. They will be displayed on this page of the JenNet-IP Browser.

The IPv6 address of each border router node is displayed on the left of the table.

To view the devices in the border router node's network tick the checkboxes on the right hand side. Then click **Discover Network(s)** to display a list of devices in the network(s).

It is possible to return to the Border Router Node Selection page by clicking **Border Routers** in the navigation bar at the top of the JenNet-IP Browser's pages. If devices are removed from or added to a network, it may be necessary to return to this page to re-discover the network and display the updated device list.

Step 3   **Display the device list**

The resulting **Network Devices** page lists the nodes in the WPAN, as illustrated in the screenshot below.
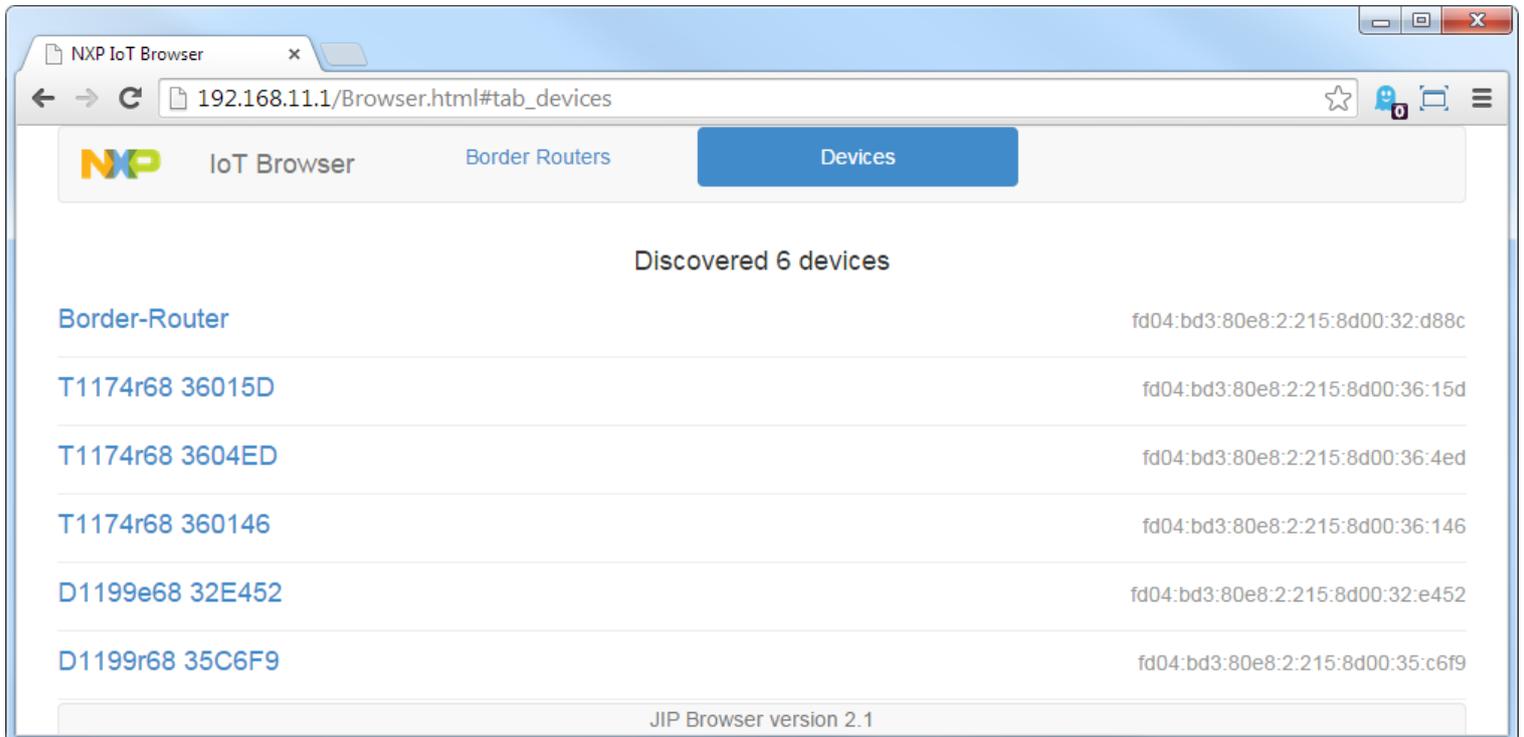
The left side displays the name assigned to the device. Devices assign themselves a name when they are started from a factory state and can be changed by the user if required. In the above example:

- '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. The default names assigned to devices use the following pattern:

  - The first character indicates the type of device, where the **makefile** for the build assigns one. In the above example 'T' indicates a template device and 'D' indicates a digital I/O device.

  - The next four hexadecimal digits indicate the Product ID part of the Device ID.

  - The fifth character indicates the node type within the network. In the above example, 'r' indicates a Router node and 'e' indicates a sleeping End Device node.

  - The next two characters indicate the JN51xx chip running the software, JN5168 in the above example.

  - The remaining hexadecimal digits after the space are the least significant digits of the device's MAC address.

The right side displays the IPv6 address of the device.

It is possible to return to the Device Selection page by clicking **Devices** in the navigation bar at the top of the JenNet-IP Browser's pages.

Step 4   **Display the template device's MIB list**

Clicking on a template device's name displays a MIBs page for that particular template device, containing a list of the MIBs on the node, as illustrated in the screenshot below:
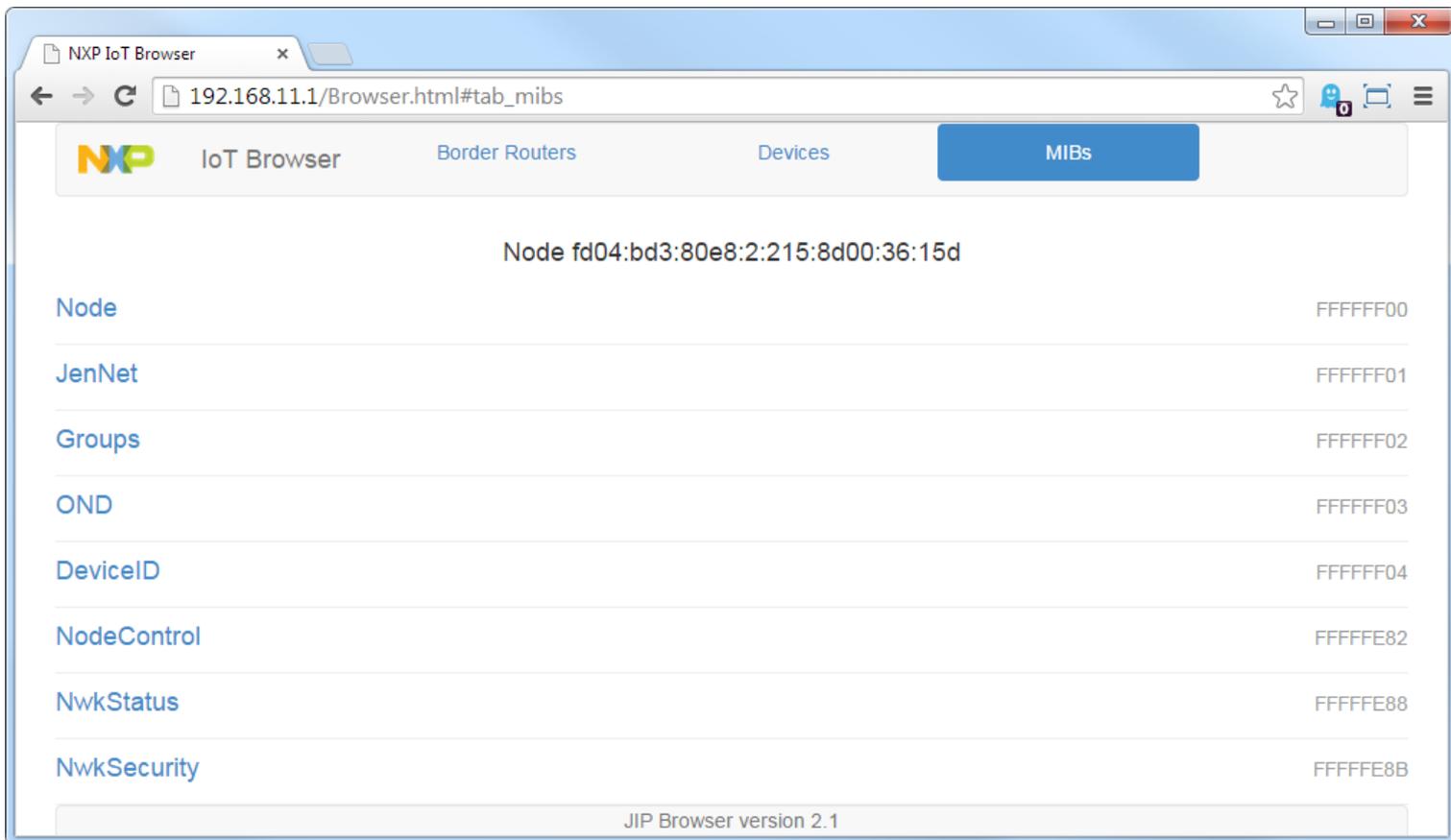
The IPv6 address of the relevant node is shown at the top of the page.

On the left side of the page are the MIB names while the MIB IDs are displayed to the right.

It is possible to return to the MIB Selection page by clicking **MIBs** in the navigation bar at the top of the JenNet-IP Browser's pages.

> ⓘ   Router nodes will respond more quickly than End Device nodes when using the JenNet-IP Browser. This is because the system may need to wait for an End Device to wake from sleep mode before communicating with it, whereas Router nodes are always powered and able to receive communications at any time.

Step 5   **Display the Node MIB's variables**

Clicking on a MIB name displays a page showing the variables within the MIB (as shown in the screenshot of the Node MIB below).
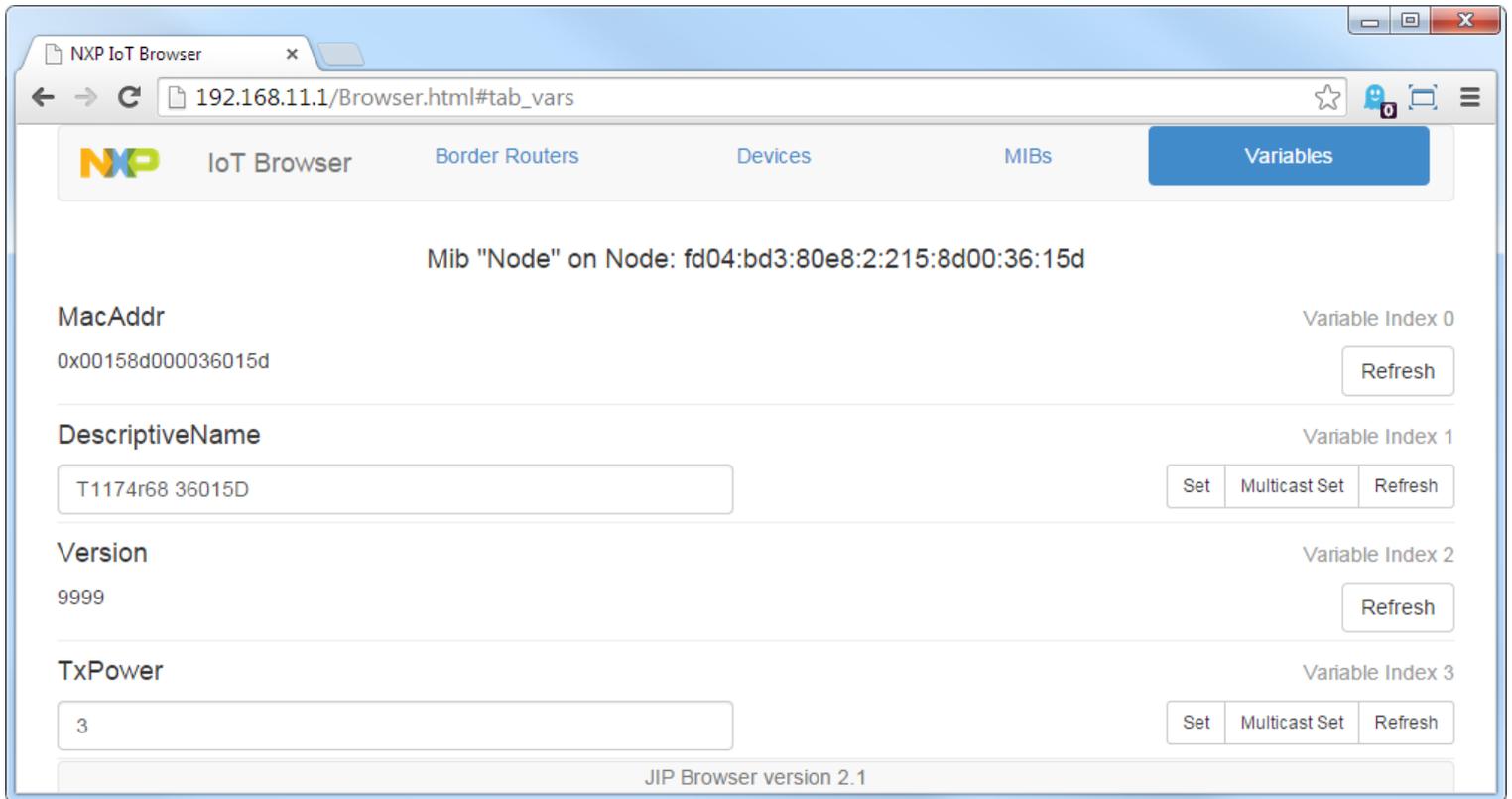
The variable names and values are shown on the left, the variable indices and control buttons are on the right.

Step 6   **Editing MIB variables**

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 **Set** button to transmit the new name to the device.
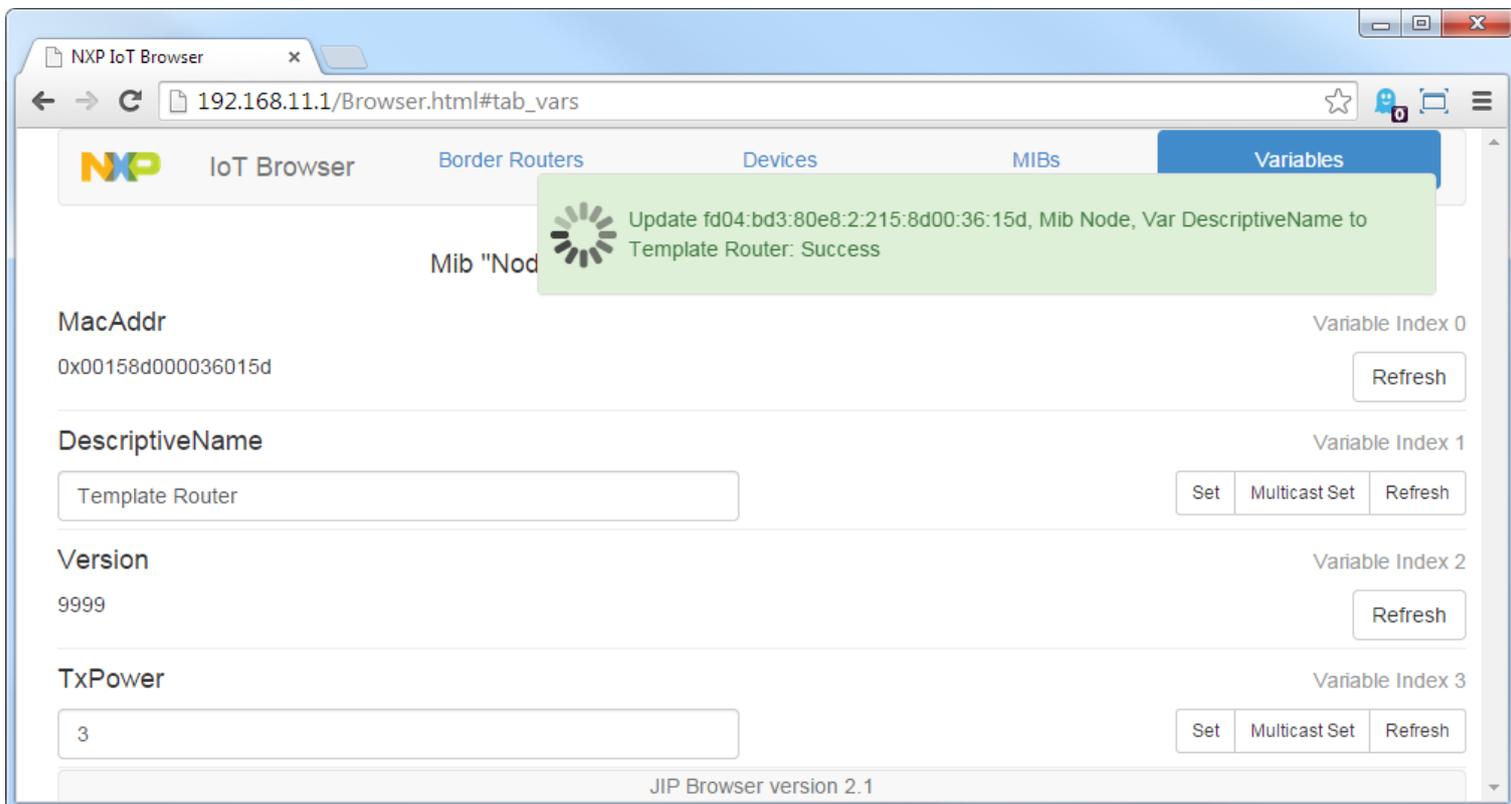
This value will be saved in permanent memory and retained even if the device is power cycled. If the network is re-discovered (by returning to the Border Routers page), it will also be displayed on the Devices list page
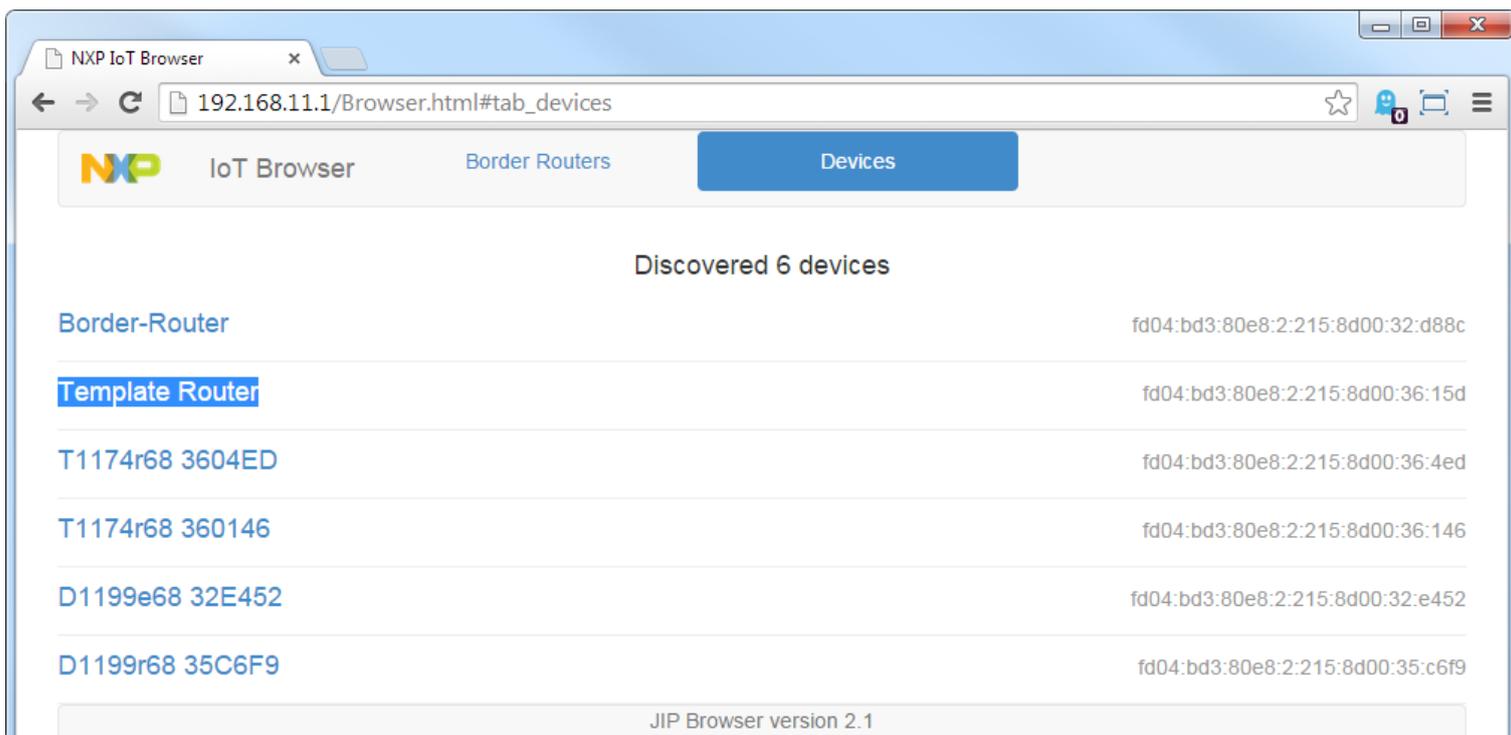
Step 7 **Reading variables**

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, as shown in the following screenshot:
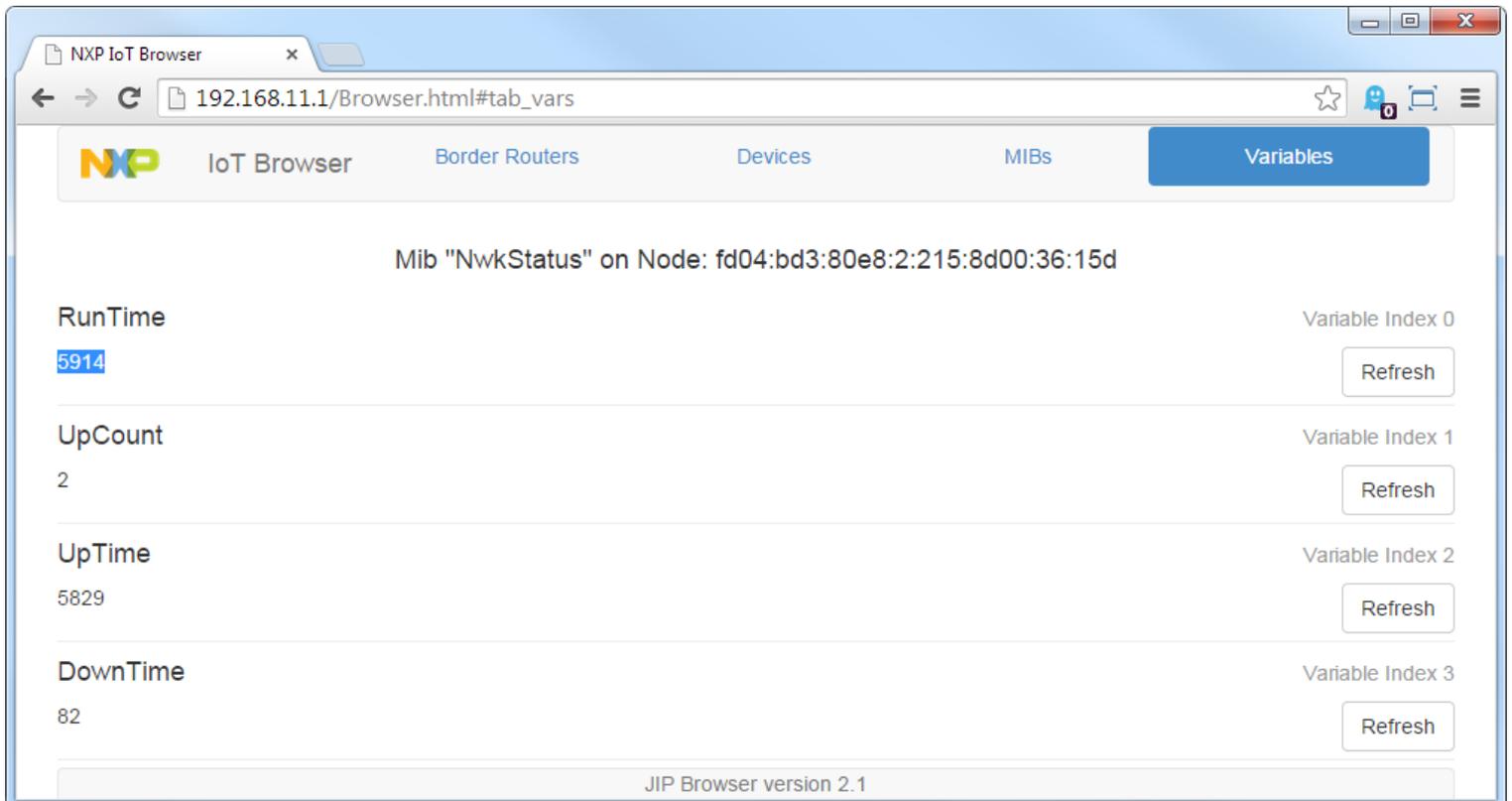
The changes in this variable can be observed by displaying and refreshing the NwkStatus MIB for a node in the browser. Individual variables can be updated by clicking the **Refresh** button to the right of the variable contents.

4.1.4 Operating the Digital I/O Devices

The digital I/O devices allow the digital I/O pins of the JN516x device to be configured, monitored and controlled remotely.

4.1.4.1 Setting Up the Digital I/O Devices

In setting up the digital I/O part of the demo system, you will need the following components:

- LAN part of the system (set up as described in Section 4.1.2.2 "Setting Up the Border Router".

- *Carrier Boards (DR1174)* fitted with JN516x modules, antennae and batteries programmed with the required firmware.

- It is recommended that the Carrier Boards are fitted with *Generic Expansion Boards (DR1199)* as these connect switches and LEDs to the input and output ins.

To set up the bulb part of the system follow the instructions below:

Step 1   **Setup digital I/O hardware**

The digital I/O device firmware can be used on any of the evaluation kit boards, as this software allows control of digital I/O. The use of the *Generic Expansion Board (DR1199)* is recommended as it provides additional buttons and LEDs to control compared to the Carrier Board alone.
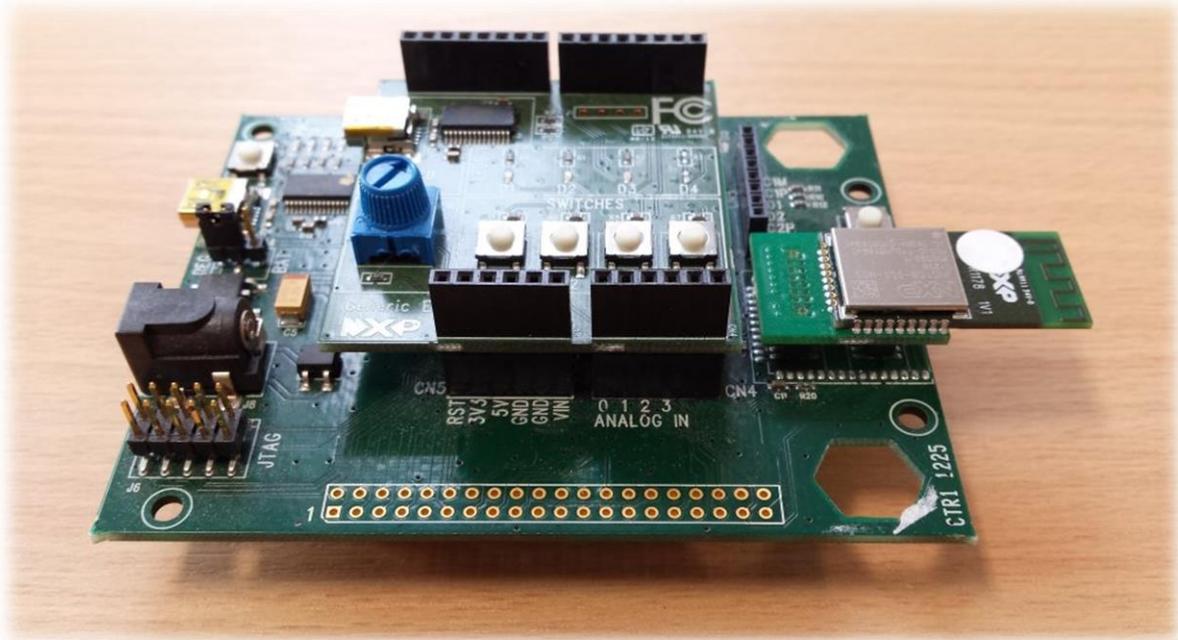
Step 2   **Program the digital I/O software**

The following pre-built binaries are provided in the Application Note for use on the evaluation kit boards.

0x11111111s_DeviceDio_DR1199_EndDevice_JN5168_v*0000*.bin
0x11111111s_DeviceDio_DR1199_Router_JN5168_v*0000*.bin

When adding these devices to a gateway system the Router or End Device may be used. The Router build will provide respond more quickly to commands compared to the End Device as the End Device can only receive messages when it wakes from sleep mode.

> **!** *It is recommended to erase the contents of the device's EEPROM when programming the device otherwise it may retain settings for an old network and be prevented from joining the new network created in this section.*

Step 5   **Add the digital I/O devices to the network**

Follow the general procedure for powering on the boards and white listing the devices described in Section 4.1.2.3 "Adding Devices to the WPAN".

On power-up, the node will attempt to join the WPAN (for which the USB dongle is the Coordinator).

Step 6   **Digital I/O device feedback**

The digital I/O device does not provide any feedback to indicate it has joined the network. The following section describes how to access the template device from the PC which can also be used to verify it has joined.

4.1.4.2 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 following digital I/O lines available on the evaluation kit boards are shown in the table below:

Board	Usage	DIO	DIO Mask	Operation
Carrier Board (DR1174)	LED D3	DIO3	0x00008	Inverted
	LED D6	DIO2	0x00004	Inverted
	Button DIO8	DIO8	0x00100	Inverted
Generic Expansion Board (DR1199)	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
LCD Expansion Board (DR1201)	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.

Step 1 **Accessing the Digital I/O device MIBs**

From the Devices page, select one of the digital I/O devices to display its MIBs as shown in the following screenshot:
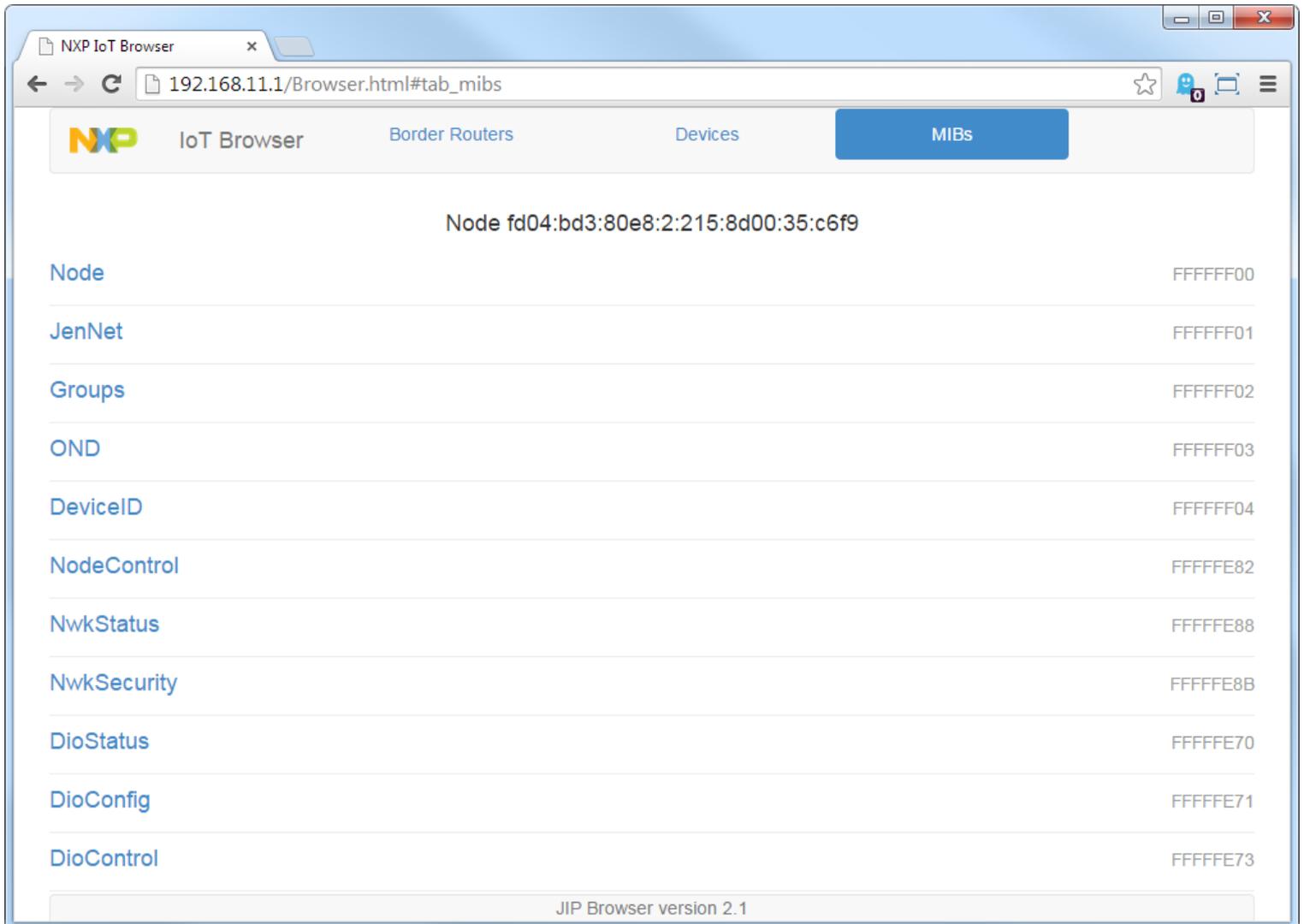
The digital I/O device contains many of the same MIBs as the template device. These MIBs present information and controls common to all device types. The digital I/O device also contain extra MIBs (where the names begin "Dio"). These MIBs expose the digital I/O functionality of the device.

Step 2   **Accessing the DioConfig MIB**

As an example, to configure the pins for the LEDs and buttons on a combination of the *Carrier Board (DR1174)* and *Generic Expansion Board (DR1199)*, navigate to the DioConfig MIB of a digital I/O device to be configured using the JenNet-IP Browser.

a)   Enter 0x21902 into the edit box for the **DirectionInput** variable and click the **Set** 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.

b)   Enter 0x1200D into the edit box for the **DirectionOutput** variable and click the **Set** 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:
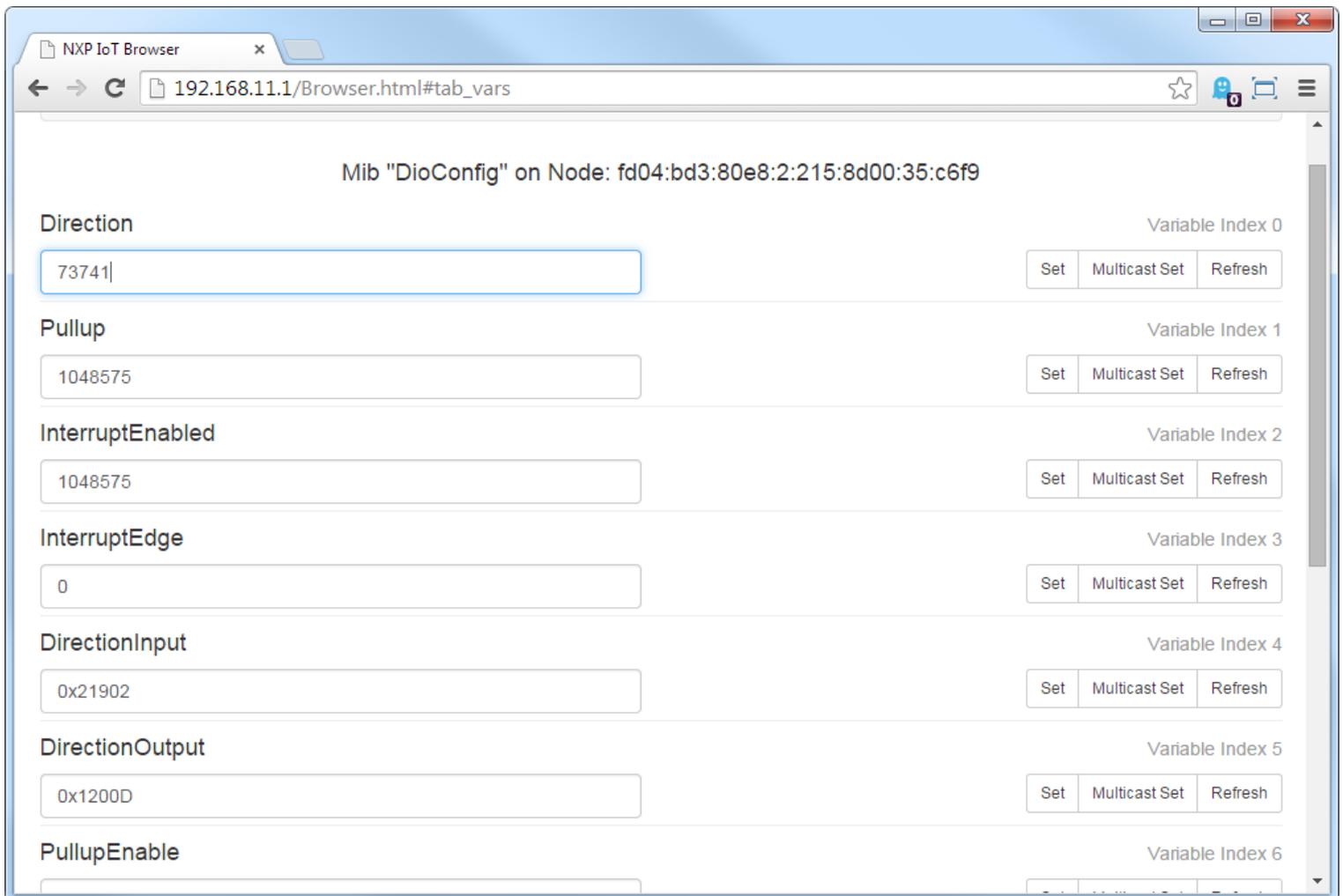
The JenNet-IP Browser displays most fields in decimal format. However, hexadecimal values may be entered by adding a leading "0x".

The **DirectionInput** and **DirectionOutput** variables operate on only a subset of the digital I/O lines. As such, when the change has been applied, the value will be reset to zero internally (refreshing these variables will demonstrate this). However, editing these values affects the **Direction** variable that can be used to set the direction of *every* digital I/O line in a single operation. Refreshing the **Direction** variable displays the updated value as shown in the above screenshot.

4.1.4.3 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.

Step 1   **Accessing the DioStatus MIB**

To read the state of the buttons simply navigate to the DioStatus MIB page for the device to monitor, as shown in the screenshot below:
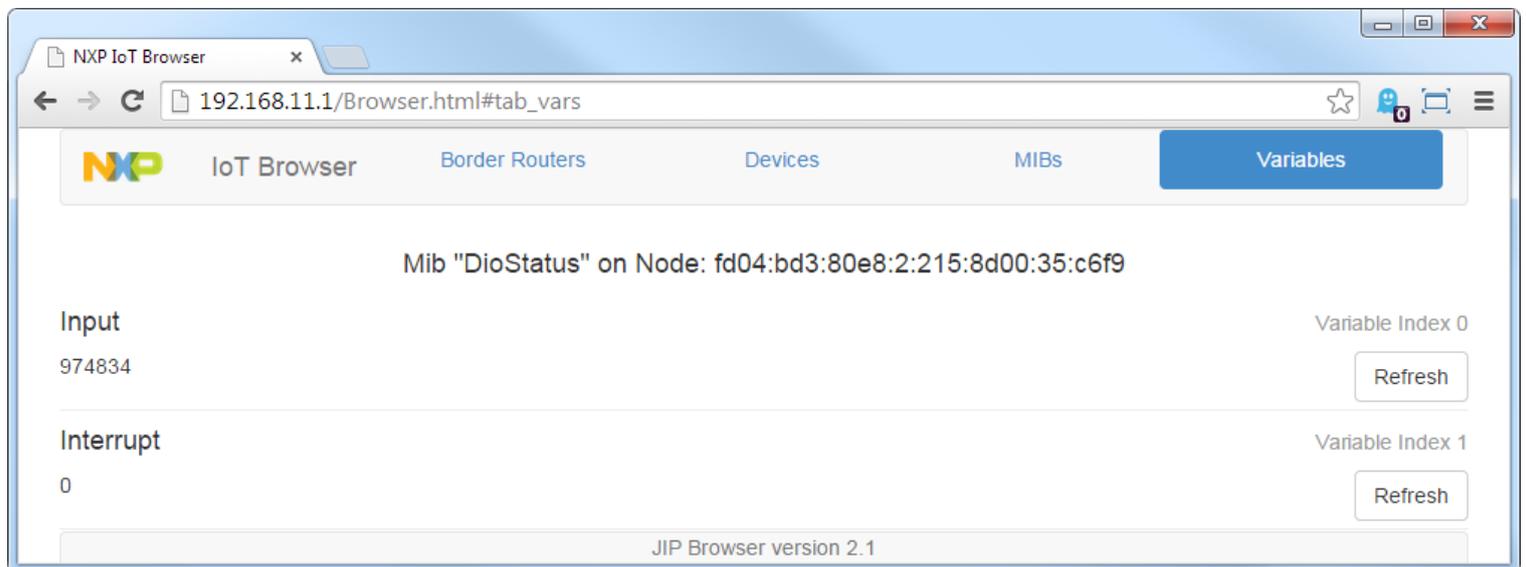
The **Input** variable indicates which inputs are currently set in the form of a bitmask. The value is displayed as a decimal number by default.

Step 2 **Changing a button state**

To check the operation of the **Input** variable, hold down a button on the evaluation kit board and refresh the browser page or **Input** variable field. The value in the **Input** variable field should change for the changed input as shown below:
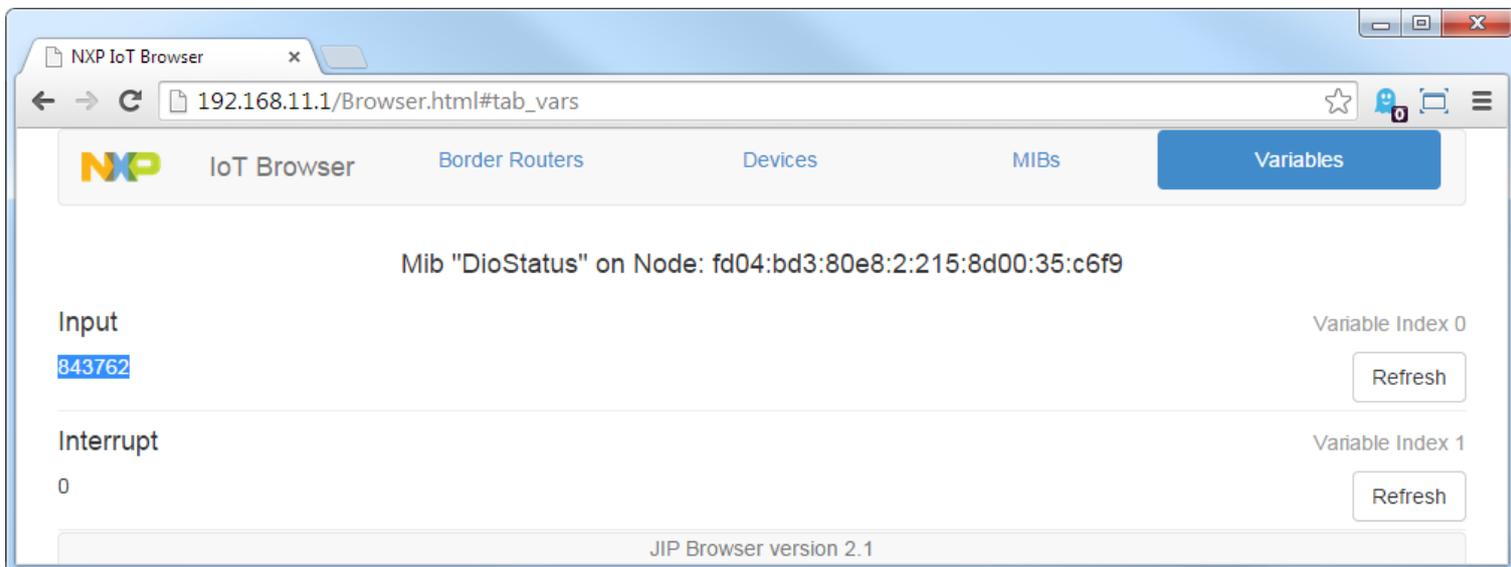
The JenNet-IP Browser displays the value in decimal notation.

The JenNet-IP Browser does not poll for new values or set up traps to monitor values, making it necessary to refresh the page or variable 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 button is pressed.

4.1.4.1 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.

Step 1 **Accessing the DioControl MIB**

To set the state of the LEDs, navigate to the DioControl MIB page for the device to control, as shown below:
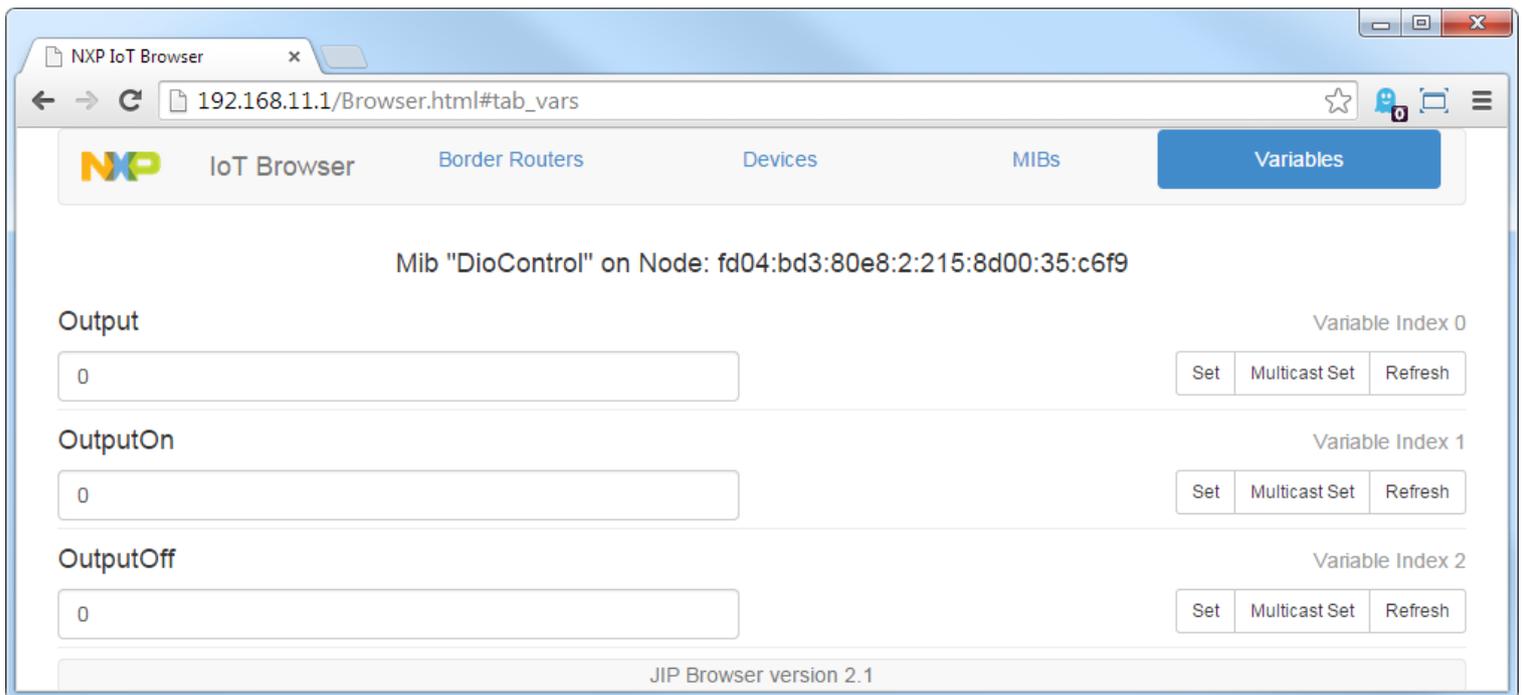
Step 2 **Output variable**

The **Output** variable indicates which outputs are currently set this information takes the form of a bitmask. The value is displayed as a decimal number, by default. Writing to the **Output** variable sets the state of *all* output pins in a single operation.

Step 3 **OutputOn and OutputOff variables**

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, leaving all other pins unchanged. Similarly, the **OutputOff** variable sets the specified output pins low, leaving all other pins unchanged.

Step 4 **Changing the LEDs**

The LEDs on the *Generic Expansion Board (DR1199)* use "Normal" operation where setting the appropriate bit high switches on the LED. While the LEDs on the *Carrier Board (DR1174)* use "Inverted" operation where setting the appropriate bit low switches on the LED.

Writing a value of 0x4 to the **OutputOff** variable will switch on LED D6 on a *Carrier Board (DR1174)*, while writing a value of 0x4 to the **OutputOn** variable will switch off LED D6, as shown below:
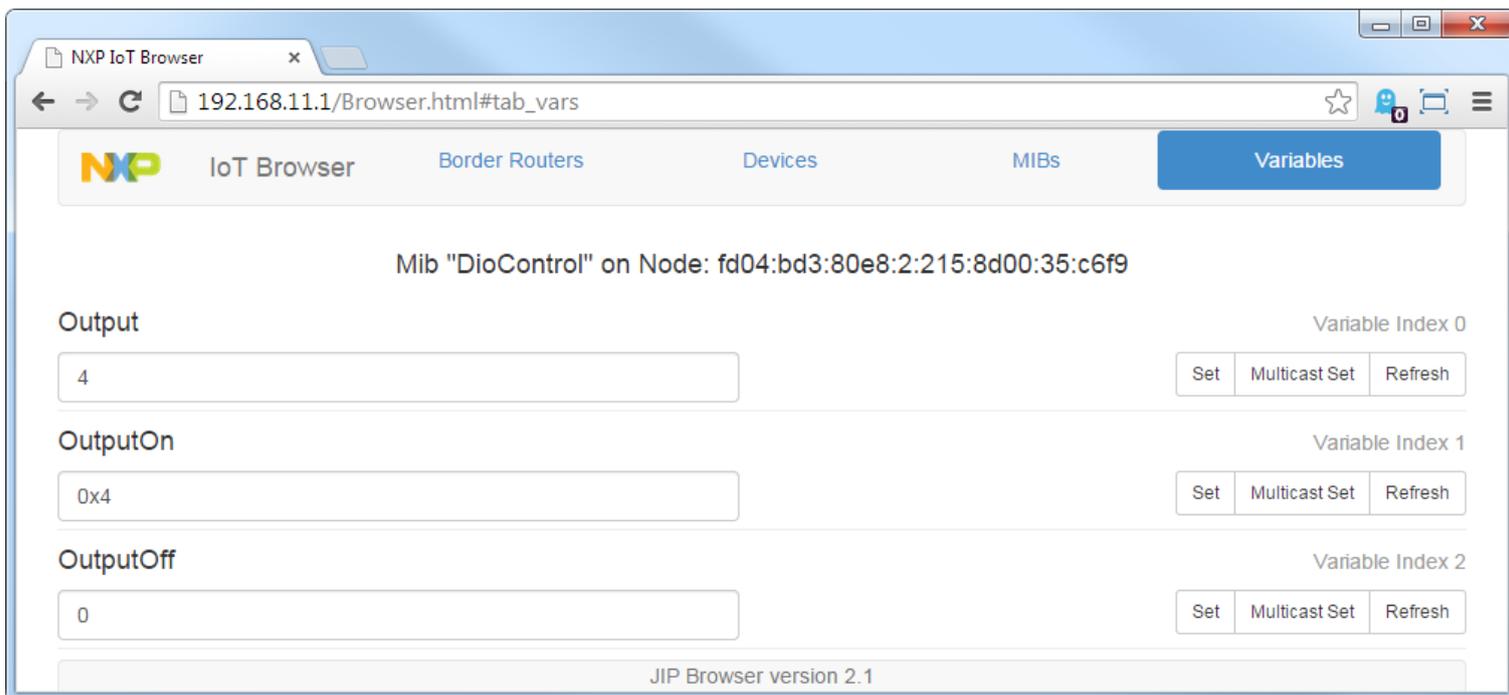
Step 5 **Checking the Output variable**

Refreshing the page or **Output** variable should display the new value resulting from any writes to the **OutputOn** or **OutputOff** variables as shown above.

The JenNet-IP Browser displays the value in decimal notation.

The JenNet-IP Browser does not poll for new values or set up traps to monitor values, making it necessary to refresh the page or variable to read back changed values.

The **OutputOn** and **OutputOff** variables are reset to zero once any writes have been applied to the **Output** variable. This can be seen by refreshing the **OutputOn** and/or **OutputOff** variables.

4.1.5 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.

> (i) End Device nodes cannot receive multicast messages, as they spend the majority of their time asleep or with the radio off.

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.

Some devices such as remote controls need to use groups unique to that device. These addresses 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.

Incorporating the MAC address of the transmitting device into the group address reduces the likelihood of a device being accidentally added to a group that can be controlled by another device. Such devices often have methods to include in their groups devices that are within direct range.

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.

A command transmitted to a group 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 and the device must be 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. Group configuration is described below.

4.1.5.1 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.

Step 1   **Accessing the Groups MIB**

To configure the node's group memberships, first navigate to the page for a Router 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.

Step 2   **View current group memberships**

The addresses of the groups to which the node currently belongs 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 the 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".

Step 3   **Add node to a group**

a)   Enter the identifier of the group in the **AddGroup** variable.

b)   Click on the **Set** button for **AddGroup**.

c)   Refresh the page or the **Groups** table variable. The new group should now appear in the **Groups** table variable.

Step 4   **Remove node from a group**

a)   Enter the identifier of the group in the **RemoveGroup** variable.

b)   Click on the **Set** button for **RemoveGroup**.

c)   Refresh the page or the **Groups** table variable. The group should now disappear from the **Groups** table variable.

Step 5   **Remove node from all groups**

a)   Enter a non-zero value in the **ClearGroups** variable.

b)   Click on the **Set** button for **ClearGroups**.

c)   Refresh the page or the **Groups** table variable. All groups should disappear from the **Groups** table variable.

4.1.5.2 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.

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.

Step 1   **Access the NodeControl MIB**

To reset a group of nodes, first navigate to the page for a Router node that is to be reset. Click on the NodeControl MIB on the Node's MIB page. This takes you to the NodeControl MIB page that lists the variables contained in the NodeControl MIB, as illustrated in the screenshot below.
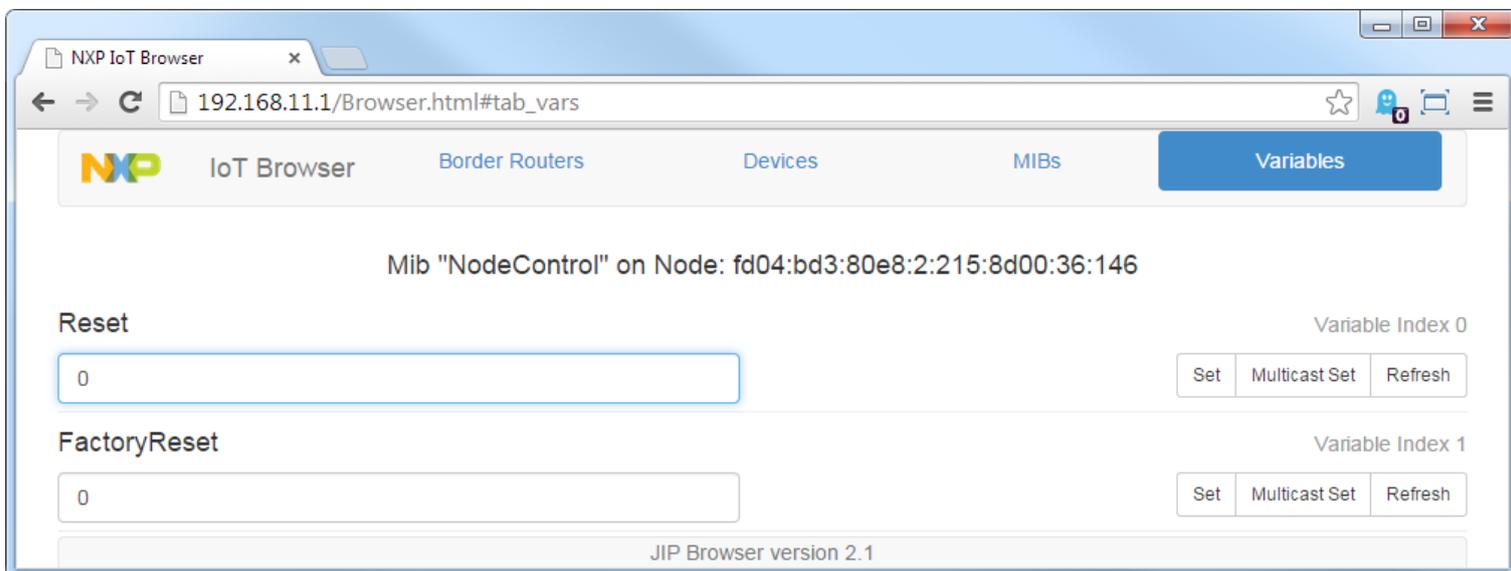
Step 2   **Set the Reset variable timer**

Writing to the **Reset** variable sets a timer, in seconds, after which the node will reset. Enter 10 to set a 10 second timer.

> ⓘ   Do not press the **Set** button or press Enter, as this will transmit a unicast message to the selected node.

Step 3 **Multicast the command to a group of nodes**

Normal updates are unicast only to the node being displayed. However, clicking the **Multicast Set** button for a variable will prompt for a multicast IPv6 address as shown below.
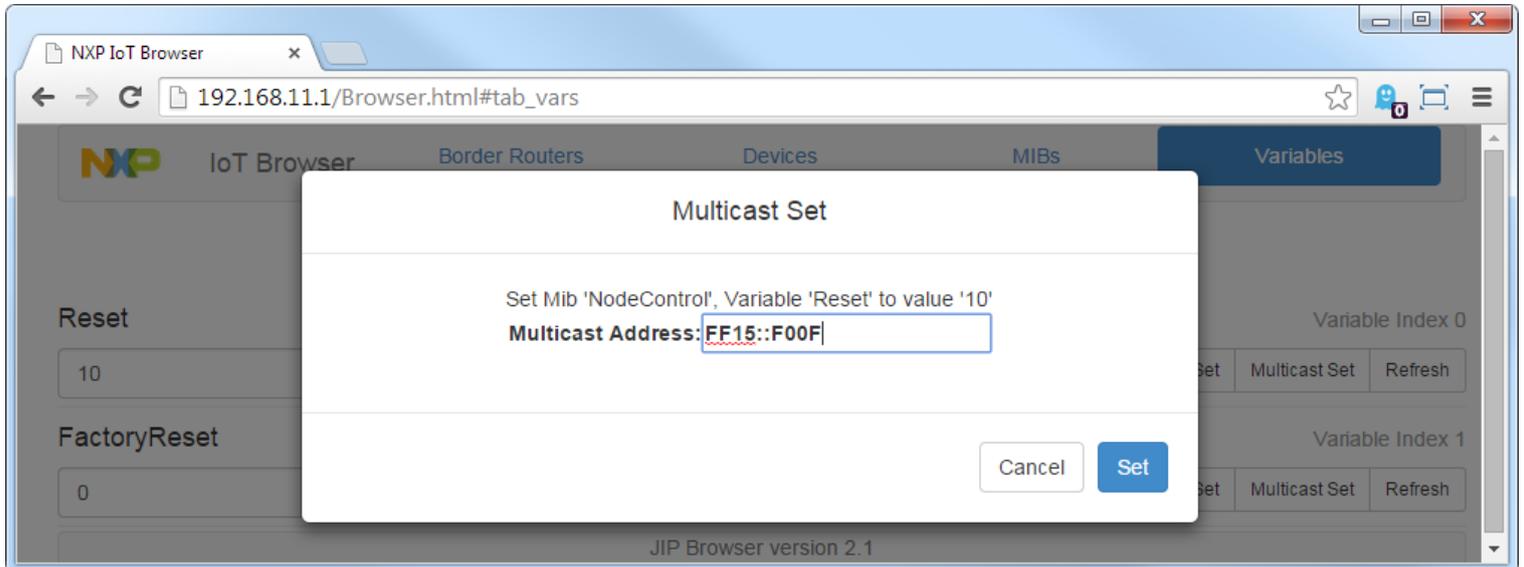
Entering the group address of FF15::F00F then clicking **Set** will transmit the command to all Router nodes that are currently members of the "All Devices" group. After 10 seconds the affected Router nodes will reset.

> ⓘ If any of the End Devices or Router nodes that are not members of the group have one of the affected Routers as their parent, they may need to re-join the network following the reset.

5 MIB Variable Reference

This section provides reference information on the MIBs and variables created by in *JenNet-IP Application Template (JN-AN-1190)*.

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 *JenNet-IP WPAN Stack User Guide (JN-UG-3080)*. The MIBs implemented by the stack are the Node, JenNet, Groups, OND and DeviceID MIBs.

5.1 Node MIBs

The Node MIBs provide access to functionality relating to node operation.

The source code for these MIBs is in the **MibCommon** folder.

5.1.1 NodeStatus MIB (0xFFFFFE80)

The NodeStatus MIB provides status information for the node.

This MIB is optional and is not included in all the example devices.

5.1.1.1 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

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

Default

Determined at power-on or reset.

Trap Notifications

None.

5.1.1.2 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.

5.1.1.3 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.

5.1.1.4 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.

5.1.1.5 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.

5.1.1.6 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.

5.1.1.7 HeapMax Variable

Description

The HeapMax variable indicates the maximum address in 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.

5.1.1.8 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 only detects the allocation of data to the stack which is written to. The current application sometimes allocates data which is not written to 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.

5.1.2 NodeControl MIB (0xFFFFFE82)

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

5.1.2.1 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

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

Default

0

Trap Notifications

On remote edits.

5.1.2.2 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

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

Default

0

Trap Notifications

On remote edits.

5.1.3 NodeConfig MIB (0xFFFFFE81)

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

5.2 Network MIBs

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

The source code for these MIBs is in the **MibCommon** folder.

5.2.1 NwkStatus MIB (0xFFFFFE88)

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

5.2.1.1 RunTime Variable

Description

The RunTime 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.

5.2.1.2 UpCount Variable

Description

The UpCount 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.

5.2.1.3 UpTime Variable

Description

The UpTime 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.

5.2.1.4 DownTime Variable

Description

The DownTime 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.

5.2.2 NwkSecurity MIB (0xFFFFFE8B)

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

5.2.2.1 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.

5.2.2.2 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.

5.2.2.3 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.

5.2.2.4 Channel

Description

The Channel variable contains the number of 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

5.2.2.5 PanId

Description

The PanId variable contains the PAN ID of the network 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

5.2.2.6 Rejoin Variable

Description

The Rejoin variable causes the device to count down 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

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

Default

0

Trap Notifications

On remote edits.

5.2.3 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.

5.2.3.1 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.

5.2.3.2 TxReq Variable

Description

The TxReq variable contains the number of test transmission attempts to be 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.

5.2.3.3 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.

5.2.3.4 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.

5.2.3.5 RxLqiMin Variable

Description

The RxLqiMin variable contains the lowest Link Quality Indicator (LQI) from the successfully received test transmission responses.

Storage

Volatile

Type

Uint8   Unsigned Integer, 8 bits

Access

Read

Values

0 to 255   Minimum LQI value of test responses.

Default

0

Trap Notifications

On remote edits.

5.2.3.6 RxLqiMax Variable

Description

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

Storage

Volatile

Type

Uint8	Unsigned Integer, 8 bits

Access

Read

Values

0 to 255	Maximum LQI value of test responses.

Default

0

Trap Notifications

On remote edits.

5.2.3.7 RxLqiMean Variable

Description

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

Storage

Volatile

Type

Uint8	Unsigned Integer, 8 bits

Access

Read

Values

0 to 255	Mean average LQI value of test responses.

Default

0

Trap Notifications

On remote edits.

5.2.3.8 CwChannel Variable

Description

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.

5.2.3.9 MacRetries Variable

Description

The MacRetries variable specifies the maximum number of MAC level retries that should be made 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.

5.2.3.10 TxLqiMin Variable

Description

The TxLqiMin variable contains the lowest LQI value 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 value of test transmissions.

Default

0

Trap Notifications

On remote edits.

5.2.3.11 TxLqiMax Variable

Description

The TxLqiMax variable contains the highest LQI value 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 value of test transmissions.

Default

0

Trap Notifications

On remote edits.

5.2.3.12 TxLqiMean Variable

Description

The TxLqiMean variable contains the mean average LQI value 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 value of test transmissions.

Default

0

Trap Notifications

On remote edits.

5.2.3.13 RxLqi Variable

Description

The RxLqi variable contains the LQI value 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 value of last received packet.

Default

0

Trap Notifications

On remote edits.

5.2.4 NwkConfig MIB (0xFFFFFE89)

This MIB is no longer included in the JenNet-IP Application Notes. It originally contained variables allowing the network operation of the device to be configured.

5.2.5 NwkControl MIB (0xFFFFFE8A)

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

5.2.6 NwkProfile MIB (0xFFFFFE8D)

This MIB is no longer included in the JenNet-IP Application Notes. It originally contained variables allowing the network profile used by the device to be configured.

5.3 Peripheral MIBs

The Peripheral MIBs provide generic information for peripheral devices.

The source code for the AdcStatus MIB is in the **MibCommon** folder.

The source code for the DIO MIBs is in **MibDio** folder.

5.3.1 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.

When the on-chip temperature ADC input is monitored the software in this MIB re-calibrates the radio and applies oscillator settings to ensure continued radio operation even when the temperature is changing.

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

5.3.1.1 Mask Variable

Description

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

Storage

Volatile

Type

| *Uint8* | Unsigned Integer, 8 bits |

Access

Read

Values

0x01	ADC 1
0x02	ADC 2
0x04	ADC 3
0x08	ADC 4
0x10	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.
0x20	ADC On-Chip Voltage

Default

Set by application

Trap Notifications

None

5.3.1.2 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.

5.3.1.3 ChipTemp Variable

Description

The ChipTemp variable contains the on-chip temperature in tenths of a degree Celsius 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.

5.3.1.4 CalTemp Variable

Description

The CalTemp variable contains the on-chip temperature in tenths of a degree Celcius 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.

5.3.1.5 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.

5.3.2 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 (e.g. bit 4 corresponds to DIO4). To manipulate the DIOs, the appropriate variable bits should be manipulated.

Each variable and its use are described in the following sub-sections:

5.3.2.1 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 u32AHI_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.

5.3.2.2 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.

5.3.3 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 (e.g. bit 4 corresponds to DIO4). To manipulate the DIOs, the appropriate variable bits should be manipulated.

Each variable and its use are described in the following sub-sections:

5.3.3.1 Direction Variable

Description

The Direction variable indicates whether each DIO line is an input or an output.

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 multiple 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 and clear bits indicate they are inputs.

Default

0

Trap Notifications

Upon input DIO direction changes.

5.3.3.2 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 multiple 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 in which set bits indicate the corresponding DIOs pull-ups are enabled.

Default

0

Trap Notifications

Upon input, DIO pull-up changes.

5.3.3.3 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 multiple 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 in which set bits indicate the corresponding DIOs input should generate interrupts.

Default

0

Trap Notifications

Upon input, DIO enabled interrupt changes.

5.3.3.4 InterruptEdge Variable

Description

The InterruptEdge variable indicates the edge on which 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 multiple 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 in which set bits correspond to leading edges.

Default

0

Trap Notifications

Upon input, DIO interrupt edge changes.

5.3.3.5 DirectionInput Variable

Description

The DirectionInput variable configures individual or multiple DIO lines to operate as inputs.

Writing to this variable is equivalent to setiing the *u32Inputs* parameter when calling the **vAHI_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 in which set bits indicate the corresponding DIOs should be configured as inputs. The directions of DIOs corresponding to clear bits are not affected.

Default

0

Trap Notifications

None

5.3.3.6 DirectionOutput Variable

Description

The DirectionOutput variable configures individual or multiple DIO lines to operate as outputs.

Writing to this variable is equivalent to setting the *u32Outputs* parameter when calling the **vAHI_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 in which 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

5.3.3.7 PullupEnable Variable

Description

The PullupEnable variable enables individual or multiple DIO internal pull-up resistors.

Writing to this variable is equivalent to setting 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 in which set bits indicate the corresponding DIO pull-ups should be enabled. The pull-ups of DIOs corresponding to clear bits are not affected.

Default

0

Trap Notifications

None

5.3.3.8 PullupDisable Variable

Description

The PullupDisable variable disables individual or multiple DIO internal pull-up resistors.

Writing to this variable is equivalent to setting 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 in which 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

5.3.3.9 InterruptEnable Variable

Description

The InterruptEnable variable enables individual or multiple DIO inputs to generate interrupts.

Writing to this variable is equivalent to setting the *u32Enable* parameter when calling the **vAHI_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 in which 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

5.3.3.10 InterruptDisable Variable

Description

The InterruptDisable variable disables individual or multiple DIO inputs to generate interrupts.

Writing to this variable is equivalent to setting the *u32Disable* parameter when calling the **vAHI_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 in which 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

5.3.3.11 InterruptRising Variable

Description

The InterruptRising variable configures individual or multiple DIO inputs to generate interrupts on a rising edge.

Writing to this variable is equivalent to setting 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 in which set bits indicate the corresponding DIO input interrupts should be raised on a rising edge. The interrupt configurations of DIOs corresponding to clear bits are not affected.

Default

0

Trap Notifications

None

5.3.3.12 InterruptFalling Variable

Description

The InterruptFalling variable configures individual or multiple DIO inputs to generate interrupts on a falling edge.

Writing to this variable is equivalent to setting the *u32Falling* 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 in which set bits indicate the corresponding DIO input interrupts should be raised on a falling edge. The interrupt configurations of DIOs corresponding to clear bits are not affected.

Default

0

Trap Notifications

None

5.3.4 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 (e.g. 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 sub-sections:

5.3.4.1 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 multiple 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 in which set bits indicate the corresponding DIO outputs are high and clear bits indicate the outputs are low.

Default

0

Trap Notifications

Upon input, DIO output changes.

5.3.4.2 OutputOn Variable

Description

The OutputOn variable sets individual or multiple DIO output line states to high.

Writing to this variable is equivalent to setting 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 in which 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

5.3.4.3 OutputOff Variable

Description

The OutputOff variable sets individual or multiple DIO output line states to low.

Writing to this variable is equivalent to setting 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 in which 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

6 Software Reference

This section provides information on the software for 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 *JenNet-IP WPAN Stack User Guide (JN-UG-3080)*.

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:
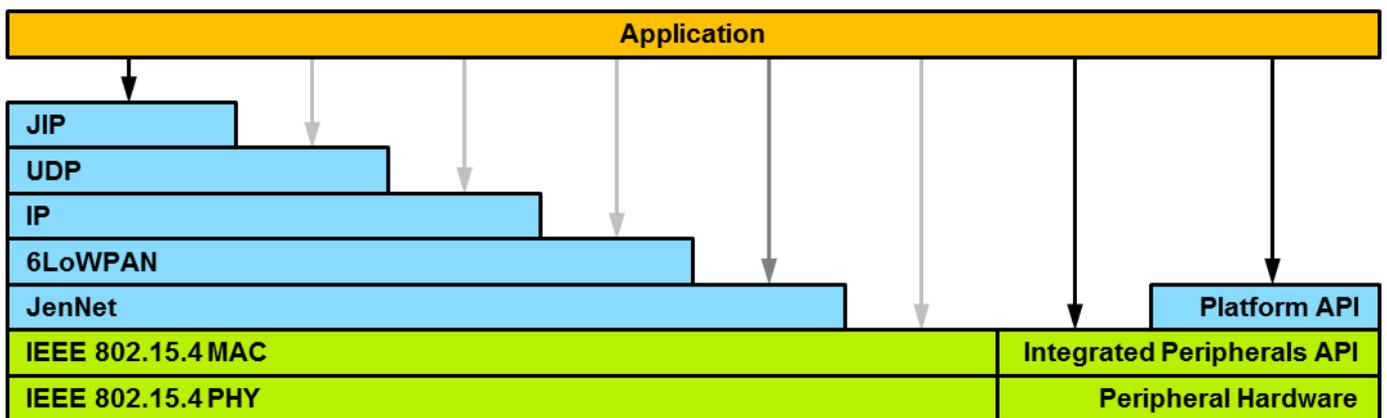
6.1 Standard Device Software Features

This section covers features that are common to all device software.

The upper layer of application software for each device group is contained in a folder named **Device***Type*, where *Type* is the type of the device. For example the upper layer of software for the template application will be found in the **Devicetemplate** folder. Within this folder the main source file for the device, including the entry point for the code, is found in a file named **Device***Type***.c**.

Every JenNet-IP device shares common functionality. This common code is located in the **Common** folder and all device types call into it to perform standard processing tasks. This code is responsible for managing the device's place in the network.

The core application functionality of a JenNet-IP device is provided by the code for the MIBs that the device supports. The code for the MIBs is grouped together where a set of MIBs provide related functionality. The code for the MIBs can be found in folders named **Mib***Group* where *Group* is the name for the group of MIBs.

The **MibCommon** folder groups together the MIBs that are found in all device types. These MIBs provide functionality to manage the node and its place in the network. The functions that implement these MIBs are called from the code located in the **Common** folder, allowing them to be easily reused in all device types.

Other **Mib***Group* folders also exist to provide MIBs that are specific to certain device types. For example, the **MibDio** folder contains the code for the digital I/O MIBs used by the digital I/O device. The functions that implement these MIBs are called directly from the **Device***Type***.c** module.

Some device types will also include a set of hardware driver source files in a folder named **Driver***Type* where *Type* is the type of driver. These drivers are called into from the MIB code that controls the hardware. All drivers of a specific type share a common interface. This makes creating different types of a particular device very easy, as only the hardware driver for the particular hardware needs to be replaced.

The image below shows these application layers for the *JenNet-IP Smart Home (JN-AN-1162)* bulb application:
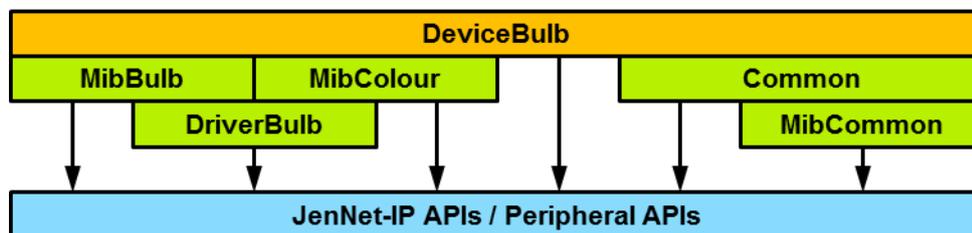
The following sections describe the features commonly found in all these components. Later sections describe the specific features of the code for each device type, the common code and the MIB code.

6.1.1 Standard Device*Type* Folder Features

The source files for each device type are found in a folder named **Device*Type***, where *Type* is the name of the device.

A **makefile** can be found in the **Build** sub-folder, while the source code is located in the **Source** sub-folder.

6.1.1.1 Standard Device*Type* Makefile

Each device type has a 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.

TARGET

This variable specifies the target name for the compilation. It should only be necessary to change this if creating a new device type from a copy of the template.

JENNIC_SDK

This variable specifies the SDK installation in the Beyond Studio for NXP toolchain that should be used to compile the application. It should not be necessary to change this value.

JENNIC_CHIP

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

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.**

JENNIC_CHIP_FAMILY

This variable specifies the chip family.

JENNIC_CHIP_SHORT

This variable is a short, two-character name for the chip and is used to form the default value for the Node MIB's Name variable.

DEVICE_NAME

This variable specifies the hardware platform for which the software should be compiled. This generally equates to the circuit board on which 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.

This value is usually 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 on the hardware platform being targeted.

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.

NODE_TYPE

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

- **Coordinator**: The node runs as a Coordinator device. Only the template device can be compiled as a Coordinator.

- **Router**: The node runs as a Router device.

- **EndDevice**: The node runs as an End Device.

This variable is usually passed into `make` on the command line.

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

NODE_TYPE_CHAR

This variable is a short, single-character name for the node type and is used to form the default value for the Node MIB's Name variable.

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. In particular, when building a system with a Coordinator based upon the template (instead of the border router) it is sensible to choose a different value for the NETWORK_ID. This is because the default Coordinator template will accept *any* node attempting to join its network and thus devices may not join the appropriate network when there is more than one in range.

CHANNEL

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

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.

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.

FACTORY_RESET_MAGIC

This variable can be used to overwrite the default magic number used to verify the EEPROM contents.

It is sometimes useful to overwrite this to force a factory reset when updating software in a device.

JENNIC_PCB

This variable specifies the evaluation kit hardware to compile for and includes the appropriate platform libraries in the compilation.

JENNIC_STACK

This variable specifies the networking stack to compile for and includes the appropriate network libraries in the compilation.

There should be no need to change this value.

JENNIC_MAC

This variable specifies the IEEE 802.15.4 MAC libraries to be compiled into the application.

There should be no need to change it from the default value.

OND_CHIPSET

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

OND_DEVICE_TYPE

This is the 32-bit ID used to identify different software builds, as used in OND.

Usually this matches JIP_DEVICE_ID but may be different where new software has a different set of MIBs from the old software, requiring a change in Device ID while still preserving the ability to update the software using OND.

TRACE

This variable when set to 1 enables debugging of application events to the UART. This adds considerable extra code.

The default value of 0 disables the debug build.

JIP_DEVICE_TYPE

This variable specifies the 16-bit Device Type ID compiled into the application.

It may be necessary to change this when creating a new device from the template.

JIP_DEVICE_TYPE_CHAR

This variable is a short, single-character name for the device type and is used to form the default value for the Node MIB's Name variable.

JIP_CR_MANUFACTURER_ID

This variable 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.

To prevent re-use of Device IDs, this value should be replaced with your own Manufacturer ID when creating your own devices.

JIP_ED_MANUFACTURER_ID

This variable 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.

To prevent re-use of Device IDs, this value should be replaced with your own Manufacturer ID when creating your own devices.

JIP_PRODUCT_ID

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

When creating your own devices you may allocate your own Product IDs when used in conjunction with your Manufacturer ID.

JIP_DEVICE_ID

This variable 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.

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.

BLD_MIB_*NAME* Variables

The set of variables beginning BLD_MIB determine which MIBs are compiled into the application. A value of 1 will build that 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.

REG_MIB *NAME* Variables

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 made 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 hardcoded values while freeing up RAM to be used for other purposes.

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 (OND) 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 OND features rely on 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.

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}**_Device*Type*_
{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* when 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.

6.1.1.2 Standard DeviceDefs.h Features

This header file contains #defines that can be used to configure the default behaviour of the device and alter the timing characteristics of the device.

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

For some devices types there may also be #defines that are accessed by the MIB modules. Such #defines must be present to allow compilation of the MIB modules.

A set of debug flags are then included that control which modules output debugging messages when debugging is enabled.

6.1.1.3 Standard Device*Type*.c Features

The main module for each device type is named **Device*Type*.c** where *Type* indicates the type of the device. All device types follow the same basic pattern described below. Where additional functionality is included, this is described in the sections for the source code for the individual device.

The standard JIP callback 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 used by the application. As such, different device types are implemented by calling into a different set of MIB modules as required.

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

#defines

There are a number of local #define values 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 that 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 units of 10ms. The default value of 25 equates to 250ms, so each reading is taken 4 times per second.

Local Variables

The following local variables are used in **Device*Type*.c**

`PRIVATE bool_t bSleep;`

This variable is used by the End Device build of the application to flag when it is ready to sleep.

Public Functions

The following public functions are commonly used:

`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_vInit()**.

`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_vInit()**.

`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 common modules, to ensure they are ready to be used.

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

The next initialisation steps for a cold start are:

Calls the **Node_bTestFactoryResetEeprom()** function 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_eJipInit()** which takes care of initialising the JenNet-IP stack and begin the process of joining a network.

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()**.

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

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

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

```
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 on the template, additional MIBs may be factory reset *before* the call to **Node_vReset()** where the device is actually reset.

```
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 on the template, additional MIBs may be registered with the stack *after* the call to **Node_eJipInit()** when the stack is up and running.

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

This callback 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.

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

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

Before entering the loop, the *bSleep* variable is set to FALSE and the loop continues until this variable is set to TRUE.

Each time around the loop:

- The on-chip watchdog is restarted.

- The common modules are given the opportunity to perform main loop processing with a call to **Node_vMain().**

- If the stack is not running the main loop is allowed to exit in order to place the device into sleep mode.

- If not entering sleep the device is placed into doze mode until the next interrupt in order to preserve power.

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

When building on the application template, other modules can perform main loop processing by calling into them from here.

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

This callback function is called by the stack for data events at the 6LowPAN level. It simply calls the common **Node_v6IpDataEvent()** 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_v6IpDataEvent()**.

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

This callback 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 prepared for sleep mode when this takes place.

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

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

This callback function is called by the stack each time a peripheral raises an interrupt. This function is called from within the interrupt context. The following peripheral device enumerations are handled in this function. (Interrupts from other peripherals can be accessed here when adapting the template to create other device types):

E_AHI_DEVICE_SYSCTRL

The application uses Wake Timer 1 for general timing purposes on End Devices, as it can be used to maintain accurate timing periods. This Wake Timer is regularly calibrated against other system clocks to maintain accuracy

On sleeping End Device the stack uses Wake Timer 0 to time sleep periods. While in a network, the stack's use of Wake Timer 0 is largely circumvented by the use of Wake Timer 1, but it is allowed to run normally when joining a network.

The wake timer interrupt events are part of the system controller and so raise interrupts for this device. These are passed on to the common modules through a call to **Node_vSysCtrlEvent()**.

E_AHI_DEVICE_TICK_TIMER

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

Coordinator and Router applications make use of this timer to maintain accurate timing periods, as this timer is always run along with the stack.

End Devices make little use of this timer as it only runs when the stack is running. It is only used to time short operations that require the use of the radio.

These events are simply passed on to the common modules through a call to **Node_vTickTimerEvent()**.

E_AHI_DEVICE_ANALOGUE

When using the common **Node.c** software the AdcStatus MIB is configured to manage the ADC peripherals to take regular readings. Each time a reading is completed an interrupt will be generated. The interrupts are passed on to **Node.c** by calling the **Node_u8AnalogueEvent()** 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 on the template).

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

This function is called by **Node_vMain()** whenever the tick timer has fired outside of interrupt context.

This function is empty in the application template but code may be added to pass tick timer events into other modules (when building on the template).

```
void Device_vAppTimer100ms ( void );
```

This function is called by **Node_vMain()** whenever the Wake Timer 1 has fired outside of interrupt context. This timer is run with an interval of 100ms by the application.

This function is empty in the application template but code may be added to pass these events into other modules when building upon the template.

```
void Device_vSecond ( void );
```

This function is called by **Node_vMain()** each time a second passes.

This function is empty in the application template but code may be added to pass these events into other modules (when building upon the template).

```
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.

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

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

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

When building on the template, additional pre-sleep handling can be added before the call to **Node_vSleep()**.

```
void Device_vPreSleepCallback ( void );
```

This function is called from **Node.c** just before the stack enters sleep mode.

The function is empty in the template application but it is a useful place to disable peripherals and external devices to preserve power while sleeping (when building upon the template).

6.1.2 Common Module Features

The **Device*Type*.c** files rely heavily on a set of common modules located in the **Common** folder.

The most important file of these is the **Node.c** file that implements code common to all node types, wrapping the use of the common MIBs into a single source file.

A detailed description of the common modules is included in *JenNet-IP Application Template (JN-AN-1190).*

6.1.3 Standard MIB Module Features

Each MIB implemented in the Application Note is included in a group with a folder named **Mib*Group*,** where *Group* defines the group name. Each MIB in a group works with the others to provide a set functionality serving a common purpose. For example the **MibCommon** folder contains MIBs that are useful in all devices types.

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

6.1.3.1 Mib*Group*.h

Mib*Group*.h contains defines used throughout the **Mib*Group*** modules. These are mostly MIB ID numbers, variable indices within each MIB and specific MIB variable values where a set of enumerations is used.

6.1.3.2 Mib*Name*Def.h

Each MIB has a MIB definition header file named **Mib*Name*Def.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.

6.1.3.3 Mib*Name*Dec.c

Each MIB has a MIB declaration file named **Mib*Name*Dec.c**. These source files make use of JenNet-IP macro definitions to declare each MIB and its variables, including 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.

6.1.3.4 Mib*Name*.h

Each MIB has a header file named **Mib*Name*.h**. This header includes the data structure definitions used by the MIB and the public function prototypes implemented by the MIB.

`typedef struct tsMibNamePerm;`

Each MIB that stores data in the PDM has a data structure named **tsMib*Name*Perm**, which is retrieved from the PDM at initialisation and stored when the data changes. The members of this structure map onto the permanent variables of the MIB.

`typedef struct tsMibNameTemp;`

Each MIB that has variables that do not need to be stored in the PDM has a data structure named **tsMib*Name*Temp**. The members of this structure map onto the temporary variables of the MIB.

```
typedef struct tsMibName;
```

Each MIB has a structure named **tsMib*Name*** that contains all the global data used by the MIB. This includes instances of the permanent and temporary data structures. This structure also includes the MIB handle, the PDM record descriptor (in MIBs that use the PDM) and other data specific to the MIB.

6.1.3.5 Mib*Name*.c

Each MIB has a source file named **Mib*Name*.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 initialisation function that is called when a device using that MIB is started. However each MIB will initialise its own data and hardware (that will vary from MIB to MIB).

The following functions are commonly found in the MIB source files, not that not all MIB implement all functions:

```
PUBLIC void MibName_vInit( thJIP_Mib  *hMibNameInit,
                           tsMibName *psMibNameInit );
```

This function initialises the MIB's data structure, reading data from the PDM if required.

```
PUBLIC void MibName_vRegister ( void );
```

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

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

```
PUBLIC void MibName_vMain ( void );
```

This function is called each time around the main loop and allows the MIB to perform frequently required processing activities.

```
PUBLIC void MibName_vTick ( void );
```

This function should be called each time the stack's 10ms tick timer fires and may be used for timing purposes by the MIB software.

The function checks if any of its MIB variables have trap updates to transmit and calls **Node_vJipNotifyChanged()** to produce the transmission.

```
PUBLIC void MibName_vAppTimer100ms ( void );
```

This function should be called each time the application's 100ms timer fires and may be used for timing purposes by the MIB software.

```
PUBLIC void MibName_vSecond ( void );
```

This function should be called once per second and may be used for timing purposes.

It is common to make a call to **Mib*Name*_vSaveRecord()** to save data to the PDM, if required.

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

This function is used to track if the node is a member of a network.

```
PUBLIC void MibName_vSaveRecord ( void );
```

This function checks the save record flag and saves the record to PDM where appropriate.

6.2 DeviceTemplate Folder

The **DeviceTemplate** folder of the Application Note contains source code for a template device. The template device can operate as a Coordinator, Router or End Device node:

- The Coordinator build is able to create the network and automatically accept Routers and End Devices as children. The Coordinator build should only be used for creating closed systems. For systems that require the use of a connection to LAN / WAN or to access devices using the JenNet-IP Browser, the border router should be used as the network Coordinator. As the Coordinator needs to be able to route messages to other nodes at any time it must be permanently powered and the radio left on in receive mode at all times.

- The Router build is able to join a network and maintain its place within the network. It can also extend the network by allowing other Routers or End Devices to join. The Routers are able to transmit and receive both unicast and multicast messages. As the Routers need to be able to route messages to other nodes at any time they must be permanently powered and the radio left on in receive mode at all times.

- The End Device build is able to join a network and maintain its place within the network. The End Devices can transmit and receive unicast messages, but they can only transmit multicast messages. The End Devices spend the majority of their time asleep and so can be battery powered. They are not able to route messages to other nodes.

The template devices contain only the core functionality of a JenNet-IP device. They do not provide any application-specific functions. Adding code to the template devices 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 devices in *JenNet-IP Smart Home (JN-AN-1162)* are based on this template code.

The application code is structured into layers, as shown below, with the code for each layer in a separate folder of the Application Note.
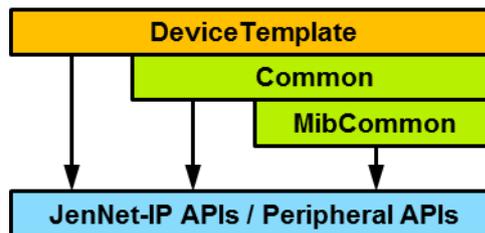
The **Common** folder contains code that is shared by all types of devices. The source file **Common\Node.c** actually provides the majority of the network joining and maintenance code. **Node.c** in turn makes use of a number of MIBs that can be used to monitor, configure and control an individual node and its operation within the network. These MIBs are implemented by the code in the **MibCommon** folder.

The source code in the **DeviceTemplate** folder contains the main module implementing the standard JIP callback functions. The callback functions then make calls into the stack libraries and common modules as required.

6.2.1 DeviceTemplate Makefile

The **makefile** for DeviceTemplate uses the variables listed in Section 6.1.1.1 "Standard Device*Type* Makefile" with the following differences:

`DEVICE_NAME`

The template device supports only the following value:

- **DR1174** for the Carrier Board (DR1174)

`NODE_TYPE`

The following builds are available for the Template device:

- **Coordinator**: The node runs as a Coordinator device.
- **Router**: The node runs as a Router device.
- **EndDevice**: The node runs as an End Device.

6.2.2 DeviceDefs.h

This header file contains #defines that can be used to configure the default behaviour of the device. These are the same as those described in Section 6.1.1.2 "Standard DeviceDefs.h Features".

6.2.3 DeviceTemplate.c

DeviceTemplate.c contains the main source code for the template application. It follows the pattern described in Section 6.1.1.3 "Standard DeviceType.c Features".

6.3 Common Folder

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.

6.3.1 Config.h

Config.h contains definitions for the default network identifiers and operating parameters that are shared by all device types.

6.3.2 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 Coordinator, 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** modules. 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. Functions called during initialisation of the device are mostly presented in the order in which they are called, although it is not a fully linear sequence.

> (i) We recommend that developers do not add functionality for device specific MIBs to this source code module. The correct place for calls into device specific MIBs is in the **Device*Name*.c** modules (refer to **DeviceDio.c** for examples of device specific MIB handling).
>
> The MIBs that are managed by **Node.c** are common to *all* devices in a system.

6.3.2.1 #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 FACTORY_RESET_MAGIC 0xFA5E13CB
```

This #define is used to verify the contents of the EEPROM 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.

```
#define NODE_WAKE_TIMER_100MS
```

This #define is used to determine if Wake Timer 1 should be used for general timing in the application.

- On Coordinators and Routers, this is set the FALSE resulting in the 100ms and 1s timers being derived from the 10ms tick timer run by the stack.

- On End Devices this is set to TRUE, resulting in the 100ms and 1s timers being derived from Wake Timer 1.

6.3.2.2 External Data

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

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

- **sMib*Name***: Data structure, data used by the MIB is contained in a structure of the type tsMib*Name* with the variable name **sMib*Name***, (where *Name* is the actual name of the MIB.) These data structure types are defined in the corresponding **Mib*Name*.h** include file of the Application Note, while the structure itself is declared in **the Mib*Name*Dec.c** source file of the Application Note which contains the MIB declaration.

- **hMib*Name***: MIB handle, passed to JIP functions to allow access to MIBs and variables. These are of the type thJIP_Mib and named **hMib*Name*** (where *Name* is the actual name of the MIB). The variable is actually declared in the **Mib*Name*Dec.c** source file of the Application Note.

6.3.2.3 Public Functions

The following public functions are implemented in **Node.c**:

```
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.

It also starts UART debugging when enabled, initialises the exception handler, enables the chip to run at high temperatures and initialises Wake Timer 1 (when enabled).

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

This function is 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 begins with the device having to be powered on for a period greater than two seconds. The device should then be powered on and off such that 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 and after two seconds the device will apply a factory reset. These timings are illustrated in the diagram below:
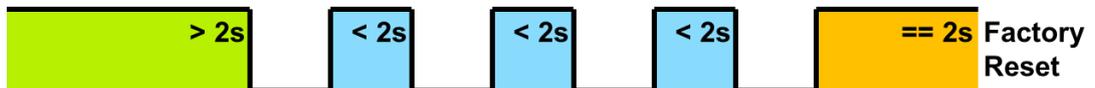
This strict pattern is used to reduce the risk of an accidental factory reset, especially if children might be playing with a wired light switch, with two long "guard" periods required either side of the three short periods.

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 that the timing sequence can be detected.

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, although 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 looking. 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 switched 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 is next switched 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 switched 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.

Two copies of the factory reset data are held in the EEPROM with the old copy only being invalidated after the new one has been written. This ensures that the loss of power during a write always leaves a valid record and avoids unintentionally performing a factory reset.

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 modules and used by the application.

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

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 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.

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 will initialise data and hardware as required.

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.

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

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

A call to **MibNwkSecurity_eJipInit()** handles the initialisation of the stack.

When running as an End Device node type, the pre-sleep callback function **Node_vPreSleepCallback()** is registered with the stack.

Once the JenNet-IP stack is running, the common MIBs used by the application are registered with the stack so that they become accessible 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.

```
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.

This function simply calls **MibNwkSecurity_v6lpConfigureNetwork()** to configure the network and apply the correct security settings for a join or re-join of the network.

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

```
void Node_v6lpResume ( void );
```

This function can be called to resume running the stack "on demand" in End Devices. This is most useful when detecting an event while awake without the stack running in order to resume the stack to transmit data.

This function simply updates the stack state. The stack is actually resumed in **Node_vMain()**.

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

This function should be called from the stack's **v6LP_DataEvent()** callback function which is called by the stack for data events at the 6LowPAN level. As this application is designed 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.

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

This function should be called from the stack's **vJIP_StackEvent()** callback 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:

STACK_STARTED, STACK_JOINED

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

The first time a device joins a network, its memberships to the multicast groups stored in EEPROM are restored.

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 cancellation 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.

STACK_RESET

Indicates the device has lost contact with its network or has been unable to join after scanning all channels.

NETWORK_ANNOUNCE

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 switch to trying to re-join a gateway network upon receiving this message, (this mode change is handled by the NwkSecurity MIB).

NODE_JOINED

This event is raised on Coordinator and Router devices when a new node is accepted as a child.

When running as a Coordinator, this clears the authorisation data currently allocated to the new child, allowing another child to join.

NODE_LEFT

This event is raised on Coordinator and Router devices when a child node is lost.

NODE_AUTHORISE

This event is raised on Coordinator devices when an unrecognised device is attempting to join the network using the older broadcast mechanisms.

In the Coordinator build, when no other device is already in the process of being authorised, the authorisation data is configured to allow the device to join the network on its next attempt.

This code could be extended to only allow new nodes to join within a certain time period of a button press or by checking a list of authorised MAC addresses (if control over which nodes are allowed to join is required).

NODE_AUTH_UCAST

This event is raised on Coordinator devices when an unrecognised device is attempting to join the network using the newer, more efficient, unicast mechanism.

This event is handled in a similar way to the NODE_AUTHORISE event in Coordinator builds.

NODE_JOINED_NWK

This event is raised on Coordinator devices when a new node is accepted into the network and it is not a direct child of the Coordinator.

When running as a Coordinator, this clears the authorisation data currently allocated to the new child, allowing another child to join.

NODE_LEFT_NWK

This event is raised on Coordinator devices when a node is lost from the network and it was not a direct child of the Coordinator.

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; NwkStatus; NwkSecurity; NwkTest.

The NwkSecurity MIB includes code to determine if the device should switch between standalone and gateway mode.

`void Node_vMain ( void );`

This function is called from the main loop in the **Device*Name*.c** file.

The **Node_vAppTimer100ms()** function is called to allow application timing every 100ms.

The **Node_vSecond()** function is called to allow application timing every second.

For End Devices, the stack state is checked and the stack is resumed when required.

The **Node_vTick()** function is called whenever the 10ms tick timer run by the stack has fired.

Finally the **vJIP_Tick()** function is called if the stack is running to provide the stack with processing time for its operations.

```
uint8 Node_vSysCtrlEvent ( 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 generates an interrupt) when a System Control peripheral interrupt is generated. This function is called from within the interrupt context.

Wake Timer 0

When the stack is run as an End Device the stack uses Wake Timer 0 to control sleep periods. Interrupts from this wake timer are used to resume the stack. This wake timer is used when joining the network. When a device is in a network, Wake Timer 1 is managed by the application to control waking and sleeping.

Wake Timer 1

The End Device builds use Wake Timer 1 running at 100ms intervals for their internal timers.

This timer is used to update the second timer that runs at a 1 second interval, allowing the application to perform timing tasks at 1 second intervals, and is picked up and processed in the main context.

Whilst running in a network as an End Device, this timer is used to resume the stack at configured intervals to maintain the device's place in the network.

The firing of the 100ms timer is also flagged for the main context to process, allowing the application to perform timing tasks at 100ms intervals.

```
uint8 Node_vTickTimerEvent ( 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 generates an interrupt) when a tick timer peripheral interrupt is generated. This function is called from within the interrupt context.

Whilst the stack is running it runs the tick timer at 10ms intervals. This function simply sets a flag which the main context reads to determine when the tick timer has fired.

On Coordinator and Router builds, this event is used to derive the 100ms and 1s timer function calls.

```
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 generates an interrupt) when an analogue peripheral interrupt is generated. 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 **MibAdcStatus_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 the ADC source of the completed reading, so that it may be passed to other modules used in the calling application.

```
void Node_vTick ( void );
```

This function is called from **Node_vMain()** every 10ms while the stack is running.

For End Devices Wake Timer 1 is checked to see if it has stalled and been restarted with a call to **Node_vCheckWakeTimer1()**.

On Coordinator builds, the timer is maintained which is used to limit the length of time an individual node may continue to attempt to join the network.

The tick is passed up to the **Device*Name*.c** module with a call to **Device_vTick(),** which in turn may pass the tick to device-specific MIBs.

Some common MIBs have timing functions that need to be called when the stack is running. These MIBs have a tick function that is called every time the tick timer has interrupted. These MIBs are the NwkStatus, NwkTest and AdcStatus MIBs.

```
void Node_vAppTimer100ms ( void );
```

This function is called from **Node_vMain()** every 100ms, derived from the Tick Timer (Coordinators and Routers) or Wake Timer 1 (End Devices).

This event is passed up to the **Device*Name*.c** file with a call to **Device_vAppTimer100ms()**, which in turn may pass the event to device-specific MIBs.

---

void Node_vSecond (void);

This function is called from **Node_vMain()** every second. Its timing is derived from the Tick Timer (Coordinators and Routers) or Wake Timer 1 (End Devices).

Where a timeout for joining a network is specified, the timer is checked here and the node prevented from joining when it expires.

When running as a Coordinator, the network announce message is broadcast every 30 seconds.

The second is passed up to the **Device*Name*.c** file with a call to **Device_vSecond()**, which in turn may pass the tick onto device-specific MIBs.

Some common MIBs have timing functions that need to be called every second. These MIBs have a second function that is called once per second. These MIBs are the NodeControl, NwkStatus and NwkSecurity MIBs.

---

void Node_vJipNotifyChanged (thJIP_Mib   hMib,
**   uint32   *pu32VarFlags,**
**   uint32   u32VarMask,**
**   uint8   u8VarCount);**

This function provides a generic way to transmit trap updates from the MIBs in the application.

Each MIB provided in the Application Notes maintains a mask of variables for which a trap update is outstanding. The MIB handle, a mask of outstanding updates, a full mask for the MIB and the number of variables in the MIB are passed into this function each time the tick timer is fired while the stack is running. The **Node_vJipNotifyChanged()** function will then transmit the trap update for a single variable, where required.

This mechanism throttles the transmission on trap updates to one per MIB every 10ms, in addition to providing a generic interface.

---

void Node_vCheckWakeTimer1 (void);

This function checks whether Wake Timer 1 has over-run and restarts it if necessary. This is used as a safety net in the event that an interrupt from Wake Timer 1 is missed.

---

uint32 Node_u32StackState (void);

This is a data access function that returns the current stack state maintained by the **Node.c** module.

---

uint32 Node_u32TimerSeconds (void);

This is a data access function that returns the current second timer value maintained by the **Node.c** module.

---

bool_t Node_bJoined (void);

This is a data access function that returns the flag that indicates if the node has joined a network since it was powered on.

`bool_t Node_bUp ( void );`

This is a data access function that returns the flag which updates if the node is currently a member of a network.

`uint64 Node_u64WakeTimer1Period ( void );`

This is a data access function that returns the timer period being used by Wake Timer 1.

`void vJIP_StayAwakeRequest ( void );`

This function is called by the stack on End Devices when a packet is received that indicates the node should stay awake to receive another packet.

In practice, the End Device actually returns to sleep but resumes the stack to receive the following packet a short time later, thus extending battery life.

`void Node_vSleep ( void );`

This function is called to place an End Device into sleep mode.

If the stack is running the node is placed into sleep mode by calling **v6LP_Sleep()** which allows the stack to complete its processing before sleeping.

If the stack is not running, Wake Timer 0 is checked to see if it has over-run and is restarted (if necessary). The chip is placed directly into sleep mode through a call to **vAHI_Sleep()**.

`void Node_vPreSleepCallback ( void );`

This function is called on an End Device just before the stack places the chip into sleep mode and allows the application a final chance to tidy any data before sleeping.

First the higher application level is notified that the chip is about to sleep through a call to **Device_vPreSleepCallback()**.

Wake Timer 1 is checked for an overrun and is restarted (if necessary) through a call to **Node_vCheckWakeTimer1()**.

6.3.3 AHI_EEPROM.h, AHI_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.

6.3.4 Exception.h, Exception.c

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

6.3.5 Security.h, Security.c

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

6.3.6 Address.h, Address.c

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

6.3.7 Table.h, Table.c

Implements data access functions for MIB table variables.

6.3.8 Uart.h, Uart.c

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

These functions provide a light-weight alternative to the full debugging libraries.

6.3.9 FtoA.h, FtoA.h

Implements routines to print float values to the UART.

6.3.10 Ovly.h

Contains #defines to strip out the use of the overlay functions in code shared with ZigBee applications.

6.3.11 Zcl.h

Contains common return values used in functions shared with ZigBee applications.

6.4 MibCommon Folder

The **MibCommon** folder contains modules that implement 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.

The common MIBs are controlled from the common **Node.c** module rather than directly from the **Device*Type*.c** module, allowing their easy integration into different device types.

6.4.1 MibNode

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. This code performs the task of saving and restoring the PDM data for this MIB.

6.4.1.1 Public Functions

The following public functions are implemented in **MibNode.c**:

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

This function initialises the Node MIB's data structure, reading it from the PDM (if available).

```
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 callback function, so that remote updates to the name can be saved to the PDM.

6.4.1.2 Callback Functions

The following callback functions are implemented in **MibNode.c**:

```
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 written to the PDM.

6.4.2 MibGroups

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. This code performs the task of saving and restoring the PDM data for this MIB.

6.4.2.1 Public Functions

The following public functions are implemented in **MibGroups.c**:

void MibGroup_vInit (tsMibGroup *psMibGroupInit);

This function initialises the Groups MIB's data structure, reading it from the PDM (if available).

If group data cannot be read from the PDM, the data is manipulated to add the device to the "All Devices" group.

The maximum number of supported groups is also passed to the stack.

void MibGroup_vRestore (void);

This function is called when the device joins a network. If the groups stored in the PDM have not yet been applied following a power cycle then the device is placed back into its groups.

PDM_teRecoveryState MibGroup_ePdmStateInit (void);

This function returns the PDM status of the Group record during initialisation. Some MIBs use this to place the device into a specific group, if data was not read from the PDM.

6.4.2.2 Callback Functions

The following callback functions are implemented in MibGroups:

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 writes the data to the PDM.

6.4.3 MibNodeStatus

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.

6.4.3.1 Public Functions

The following public functions are implemented in **MibNodeStatus.c**:

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

This function initialises 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.

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

This function registers the MIB with the stack, making the variables available to be accessed by other devices, and ensures updated data is written to the PDM.

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

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

6.4.4 MibNodeControl

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

6.4.4.1 Public Functions

The following public functions are implemented in **MibNodeControl.c**:

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

These functions are implemented in the standard way.

```
void MibNodeControl_vSecond ( uint32 u32TimerSeconds );
```

This function should be called once per second. It checks the timers used to schedule a reset or a factory reset. When the scheduled time is reached, it performs the appropriate reset.

```
teJIP_Status MibNodeControl_vSetReset ( uint16 u16Val
                                  void   *pvCbData );
```

This function is called by the stack whenever the Reset variable is written to. It sets the time at which the node should be reset.

```
teJIP_Status MibNodeControl_vSetFactoryReset (
                                  uint16 u16Val
                                  void   *pvCbData );
```

This function is called by the stack whenever the FactoryReset variable is written to. It sets the time at which the node should be factory reset.

6.4.5 MibNwkStatus

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 re-applied across power cycles.

6.4.5.1 Public Functions

The following public functions are implemented in **MibNwkStatus.c**:

```
void MibNwkStatus_vRegister   ( void );
void MibNwkStatus_vTick       ( void );
void MibNwkStatus_vSaveRecord ( void );
```

These functions are implemented in the standard way.

```
void MibNwkStatus_vInit (
                    thJIP_Mib        hMibNwkStatusInit,
                    tsMibNwkStatus *psMibNwkStatusInit,
                    bool_t          bMibNwkStatusSecurity );
```

This function initialises the NwkStatus MIB's data structure, reading it from the PDM (if available). It also initialises internal variables from the PDM data.

The frame counter read from Flash memory is increased to ensure it is greater than the value used prior to the device being powered off.

```
void MibNwkStatus_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 flagged.

The various timers used to monitor the run-time, up-time and down-time of the node in the network are updated and trap update flags are set when required.

A call is made to **MibNwkStatus_vSaveRecord()** to save data to the PDM, if required.

```
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 is set to write the data to the PDM.

6.4.6 MibNwkSecurity

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 switching between gateway and standalone modes.

6.4.6.1 Public Functions

The following public functions are implemented in **MibNwkSecurity.c**:

```
void MibNwkSecurity_vInit      ( void );
void MibNwkSecurity_vRegister ( void );
```

These functions are implemented in the standard way.

```
teJIP_Status MibNwkSecurity_eJipInit ( void );
```

This function is called from the **Node_eJipInit()** function. It sets up the stack initialisation structure passing it into **eJIP_Init()** to start the stack running.

```
void MibNwkSecurity_v6lpConfigureNetwork (
                tsNetworkConfigData *psNetworkConfigData );
```

This function is called from the **Node_v6lpConfigureNetwork()** function. It allows the NwkSecurity MIB to apply its network security data during stack configuration.

The configuration data in the *psNetworkConfigData* structure is updated to complete the stack configuration.

If the network and commissioning keys have not been retrieved from the PDM the defaults keys are calculated and saved. Only Coordinator builds construct a network key. The other devices join the network using one of their commissioning keys and obtain the network key when they successfully join.

If the device has already been in a network, the configuration data is updated to only allow it to scan on the known channel and to join the network with the known PAN ID.

If the device already has a network key, having previously been a member of a network, the network key is restored.

If the device was in a gateway system, the stack is enabled to receive broadcasts on the old channel and PAN ID while it attempts to re-join.

If the device was in a Standalone network the security descriptors for the other nodes in the network (that it is aware of) are restored and the device is placed back into standalone mode.

If the device has never been in a network and it is a Router or End Device, the gateway commissioning key is restored for use while joining. Coordinator devices simply apply the network key which they are to use for their network.

```
void MibNwkSecurity_vSetUserData ( void );
```

This function adds the Network ID and Device Type IDs of the device to the beacon response and establish route messages that the stack transmits.

```
void MibNwkSecurity_vMain ( void );
```

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

The function checks to see if a Beacon Response has been received from a node in standalone commissioning mode and completes the process of switching to standalone mode to allow it to join the standalone network.

```
void MibNwkSecurity_vSecond ( uint32 u32TimerSeconds );
```

This function should be called once per second.

The network re-join timer is checked. When the scheduled time to re-join a network is reached, 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.

```
uint8 MibNwkSecurity_u8StackEvent (
                          te6LP_StackEvent    eEvent,
                          uint8               *pu8Data,
                          uint8               u8DataLen );
```

This function should be called to pass stack events into the NwkSecurity MIB.

The following events are handled:

STACK_STARTED, STACK_JOINED

These events indicate that the node has created or joined a network.

The OND system is initiailised.

The channel and PAN ID of the network are stored. They are also applied to the network configuration data to ensure any future re-joins only scan the known channel and limit join attempts to the known PAN ID.

On Routers and End Devices, the network key is noted. If a standalone system has been joined, the node is taken out of standalone commissioning mode and the parent's security information saved.

Finally the PDM data is saved to EEPROM.

STACK_RESET

The E_STACK_RESET event is raised in two different situations:

1. 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. The device is currently in a network. It indicates that contact with the network has been lost.

Code is included to handle the E_STACK_RESET event that is expected when switching from standalone to gateway mode. This code reverts back to gateway mode in case of failure to join a standalone network.

STACK_NETWORK_ANNOUNCE

The E_STACK_NETWORK_ANNOUNCE 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.

```
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.

```
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.

```
void MibNwkSecurity_vResetSecureAddr ( void );
```

This function clears all the reserved security table entries.

```
void MibNwkSecurity_vSetSecurityKey ( uint8 u8Key );
```

This function is used to set the appropriate commissioning or network key for the current stack mode.

```
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, the new values are saved by the PDM. Note that the new security keys are not applied until the node is restarted.

```
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**.

```
teJIP_Status MibNwkSecurity_vSetRejoin ( uint16 u16Val
                              void    *pvCbData );
```

This function is called by the stack whenever the Rejoin variable is written to. It sets the time at which the node should re-join a network via the full commissioning process.

6.4.6.2 Callback Functions

The following callback functions are implemented in **MibNwkSecurity.c**:

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

This function is called by the stack whenever a Beacon Response is received.

Coordinator devices simply accept any Beacon Responses, as they are used only when automatically selecting a PAN ID which must be different from any other PAN IDs being used in the vicinity.

In Router and End Device builds:

- The Network ID included in the response is validated against the Network ID of the WPAN the device is trying to join.

- The LQI level of the response is checked to ensure it meets minimum requirements. The Join Profile being used by the stack may impose higher requirements.

- Where a Beacon Response is received from a node in standalone commissioning mode, the node begins the process of switching to standalone mode.

```
bool_t MibNwkSecurity_bScanSortCallback (
                    tsScanElement  *pasScanResult,
                    uint8          u8ScanListSize,
                    uint8          *pau8ScanListOrder );
```

This function provides an alternative parent sorting algorithm to that used by the stack. The callback is disabled by default but can enabled by removing the comment around the call to **vApi_RegScanSortCallback()** in the **MibNwkSecurity_vSetUserData()** function.

The stack's ordering for Routers trying to join a network is:

1. Depth – closest to the Coordinator, to minimise packet hops through the network.

2. Children – fewest children, to spread the work of routing packets through the network among the Routers.

3. Signal Strength – highest to ensure the best connection (a minimum signal strength is also imposed by the stack).

The stack's ordering for End Devices trying to join a network is:

1. Signal Strength – highest to ensure the best connection (a minimum signal strength is also imposed by the stack).

2. Children – fewest children, to spread the work of routing packets through the network among the Routers.

3. Depth – closest to the Coordinator, to minimise packet hops through the network.

The default sorting algorithm works well in most situations but Routers in particular may choose a relatively weak link in order to join as close to the Coordinator as possible. This alternative sorting algorithm attempts to balance the need to join as high up the tree as possible while still maintaining a strong link to the chosen parent.

The alternative algorithm makes use of a pivot signal strength set by the #define MIB_NWK_SECURITY_SCAN_SORT_PIVOT_LQI that has a default value of 96.

Responses with signal strengths at or above the pivot LQI are sorted above those below the pivot LQI.

The results at or above the pivot LQI are then sorted by Depth, Children then LQI (as for Routers above). This ensures that the results with a relatively strong signal still favour joining close to the Coordinator.

The result below the pivot LQI are then sorted by LQI, Children then Depth (as for End Devices above). This ensures that the results with a relatively weak signal favour joining the strongest links.

```
bool_t MibNwkSecurity_bScanSortCheckSwap (
                        tsScanElement    *pasScanResult,
                        uint8             u8ScanListItem,
                        uint8            *pau8ScanListOrder );
```

This is the sorting function for the pivot based scan result ordering. It compares two entries in the results list, swapping their positions if necessary.

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

This function is called by the stack whenever a node is trying to join the network and establish its route to the Coordinator.

The Network ID of the node trying to join is validated against the Network ID of the potential parent node.

6.4.7 MibNwkTest

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.

6.4.7.1 Public Functions

The following public functions are implemented in **MibNwkTest.c**:

```
void MibNwkTest_vInit      ( thJIP_Mib      hMibNwkTestInit,
                             tsMibNwkTest *psMibNwkTestInit );
void MibNwkTest_vRegister ( void );
```

These functions are implemented in the standard way.

```
void MibNwkTest_vTick ( void );
```

This function should be called every 10ms to allow the NwkTest MIB to time the transmission of its test messages.

```
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 as a destination address for test messages while the test is running.

```
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.

6.4.7.2 Callback Functions

These callback functions are implemented in **MibNwkTest.c**:

```
void vJIP_Remote_DataSent ( ts6LP_SockAddr *psAddr,
                            teJIP_Status    eStatus );
```

The stack calls this callback function to indicate the status of a transmission attempt. While the test is running, a successful transmission increments the TxOk variable.

```
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 value of the received packet measured and used to update the RxLqiMin, RxLqiMax and RxLqiMean MIB variables.

A successful response contains the LQI value of the transmitted test packet, allowing the **TxLqiMin**, **TxLqiMax** and **TxLqiMean** values to be calculated.

6.4.8 MibAdcStatus

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.

6.4.8.1 Public Functions

The AdcStatus MIB contains the following public functions:

void MibAdcStatus_vRegister (void);

This function is implemented in the standard way.

void MibAdcStatus_vInit (thJIP_Mib   hMibAdcStatusInit,
**   tsMibAdcStatus *psMibAdcStatusInit,**
**   uint8   u8AdcMask,**
**   uint8   u8Period);**

This function is called from the **MibAdcStatus_vInit()** patch function.

It performs basic initialisation of the MIB's data. The ADC readings are initialised with a call to **MibAdcStatus_vResume()**.

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 using the **MibAdcStatus_vStart()** function.

void MibAdcStatus_vResume (void);

This function starts or resumes running the ADC conversions following a cold or warm start initiating the **MibAdcStatus_vStart()** function.

void MibAdcStatus_vStart (void);

This function begins the reading of the next ADC input.

```
uint8 MibAdcStatus_u8Analogue ( void );
```

This function should be called each time an analogue interrupt is generated 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.

When the ADC result is from the on-chip temperature sensor, it is converted to tenths of a degree Celsius. 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 action is taken if required.

This function returns the ADC source of the reading, allowing other interested source code modules to be updated with the new reading.

```
uint16 MibAdcStatus_u16Read ( uint8 u8Adc );
```

This function returns the most recent raw 12-bit reading for the specified ADC source.

```
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.

```
int16 MibAdcStatus_i16DeciCentigrade ( uint8 u8Adc );
```

This function converts and returns the most recent reading for the specified ADC as a temperature in tenths of a degree Celsius.

6.5 DeviceDio Folder

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 Section 6.1 "Standard Device Software Features". The additional functionality is contained within the DIO MIB source code, described later. 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:
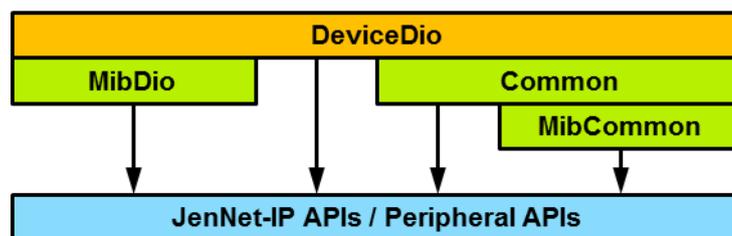
The code to implement the DIO MIBs is contained in **MibDio** folder. The DIO device also makes use of the common code and common MIBs in the **Common** and **MibCommon** folders.

6.5.1 DeviceDio Makefile

The **makefile** for DeviceDio uses the variables listed in Section 6.1.1.1 "Standard Device*Type* Makefile" with the following differences:

`DEVICE_NAME`

The DIO device supports only the following value:

- **DR1199** for the Generic Expansion Board (DR1199).

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.

`NODE_TYPE`

The following builds are available for the DIO device:

- **Router**: The node runs as a Router device.
- **EndDevice**: The node runs as an End Device.

`BLD_MIB_NAME Variables`

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

`REG_MIB_NAME Variables`

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

6.5.2 DeviceDefs.h

This header file contains a few #defines that can be used to configure the default behaviour of the device. These are the same as those described in Section 6.1.1.2 "Standard DeviceDefs.h Features".

Additional debug flags for DIO MIBs are included.

6.5.3 DeviceDio.c

DeviceDio.c contains the main source code for the digital I/O application. It follows the pattern described in Section 6.1.1.3 "Standard DeviceType.c Features" with the following additions:

6.5.3.1 #includes

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

6.5.3.2 External Variables

External data variables are added to access the data and handles of the DIO MIBs. Each MIB has two variables:

- **sMib*Name***: Data used by the MIB is contained in a structure of the type tsMib*Name* with the variable name **sMib*Name***, (where *Name* is the actual name of the MIB.) These data structure types are defined in the corresponding **Mib*Name*.h** include file of the Application Note, while the structure itself is declared in **the Mib*Name*Dec.c** source file of the Application Note which contains the MIB declaration.

- **hMib*Name***: MIB handle passed to JIP functions to allow access to MIBs and variables. These are of the type thJIP_Mib and named **hMib*Name*** (where *Name* is the actual name of the MIB). The variable is actually declared in the **Mib*Name*Dec.c** source file of the Application Note.

6.5.3.3 Public Functions

The following public functions are implemented in **DeviceDio.c**:

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

The DIO MIB modules are initialised here.

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

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

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

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

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

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.

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

This function is called from the common **Node.c** module every 10ms when the stack is running.

Tick functions in the DIO MIBs are called to allow the MIBs to interact with the stack.

```
void Device_vAppTimer100ms ( void );
```

This function is called from the common **Node.c** module every 100ms.

MibDioStatus_vAppTimer100ms() is called to allow the DioStatus MIB to perform its regular processing.

6.6 MibDio Folder

The MibDio folder contains modules that implement MIBs that can be reused in many different device types. The MIBs provide access for configuring, monitoring and controling the digital I/O lines of the JN516x chips.

6.6.1 MibDioConfig

The DioConfig MIB allows the operation of the digital I/O lines to be configured. The directions, pull-ups and interrupts of the I/O lines can all be configured.

6.6.1.1 Public Functions

The following public functions are implemented by MibDioConfig:

```
void MibDioConfig_vRegister ( void );
void MibDioConfig_vTick     ( void );
```

These functions are implemented in the standard way.

```
void MibDioConfig_vInit ( thJIP_Mib        hMibDioConfigInit,
                          tsMibDioConfig *psMibDioConfigInit );
```

This function initialises 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.

```
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 );
```

These functions are called by the stack to set the value of 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.

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 );
```

These functions are called by the stack to set the value of 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.

The above functions operate by calling the Integrated Peripherals API functions. As such they may be used to configure a subset of the digital I/O pins leaving other digital I/O pins unchanged.

6.6.2 MibDioStatus

The DioStatus MIB provides information on the status of the Digital I/O lines, including the current input states and interrupt flags.

6.6.2.1 Public Functions

The following public functions are implemented by MibDioStatus:

```
void MibDioStatus_vRegister ( void );
void MibDioStatus_vTick     ( void );
```

These functions are implemented in the standard way.

```
void MibDioStatus_vInit ( thJIP_Mib         hMibDioStatusInit,
                          tsMibDioStatus *psMibDioStatusInit );
```

This function initialises the DioStatus MIB's data structure.

The initial values of the variables are read from the JN51xx registers.

```
void MibDioStatus_vAppTimer100ms ( void );
```

This function should be called every 100ms. The states of the digital I/O input lines are read from the JN516x register.

```
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.

6.6.3 DioControl MIB

The DioControl MIB allows output lines of the JN51xx to be controlled.

6.6.3.1 Public Functions

The following public functions are implemented in MibDioControl:

```
void MibDioControl_vRegister ( void );
void MibDioControl_vTick     ( void );
```

These functions are implemented in the standard way as described in
Mib*Name*.c.

```
void MibDioControl_vInit (
                          thJIP_Mib          hMibDioControlInit,
                          tsMibDioControl *psMibDioControlInit );
```

This function initialises 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.

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 );
```

This function is called by the stack when the Output variable is set remotely. It
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 );
```

These functions are called by the stack when the OutputOn and OutputOff
variables are written to. They 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.

6.7 DeviceProtocol Folder

The **DeviceProtocol** folder of the Application Note contains source code for a generic template device that allows messaging at the 6LoWPAN layer in addition to using MIBs and variables. This allows custom protocols to be used where the use of MIBs and variables is inappropriate. The protocol device can operate as a Coordinator, Router or End Device node.

The source code has been constructed by adding to the code for the template device described in Section 6.1 "Standard Device Software Features". The additional functionality is contained within the **Protocol.c** module. The code in the **DeviceProtocol.c** just makes calls to the **Protocol.c** 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 protocol device is the best way to create new types of JenNet-IP / 6LoWPAN 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 DeviceProtocol application on top of the JenNet-IP WPAN Stack:
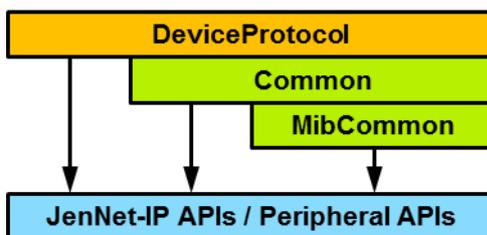
To illustrate the use of a custom protocol the device implements a simple protocol in which each node sends a Register Request command to the Coordinator every 15 seconds. The Coordinator then replies with a Register Response command.

The protocol device makes use of the common code and common MIBs in the **Common** and **MibCommon** folders.

6.7.1 DeviceProtocol Makefile

The **makefile** for DeviceProtocol uses the variables listed in Section 6.1.1.1 "Standard Device*Type* Makefile" with the following differences:

`DEVICE_NAME`

The protocol device supports only the following value:

- **DR1174** for the Carrier Board (DR1174)

`NODE_TYPE`

The following builds are available for the protocol device:

- **Coordinator**: The node runs as a Coordinator device.
- **Router**: The node runs as a Router device.
- **EndDevice**: The node runs as an End Device.

NETWORK_ID

The Network ID is set to the non-standard value of 0x00001190. This is to ensure that when building a system using the Coordinator build, the other nodes do not join a standard (MIB and variable only) network.

The devices in this system are able to join a network managed by a border router in the normal way but the Router and End Devices will need to be re-built using the standard (0x11111111) Network ID.

Devices using 6LoWPAN messaging in this way in a gateway system will only be able to communicate with other devices using 6LoWPAN messages if both the source and destination devices have the same IPv6 port open. To allow the border router to participate in such communications, you will need to add additional code to open the correct port. Coordinator and Router nodes that do not have the IPv6 port open will still route such packets through the network for the nodes that do.

6.7.2 DeviceDefs.h

This header file contains a #defines that can be used to configure the default behaviour of the device. This are the same as those described in Section 6.1.1.2 "Standard DeviceDefs.h Features", with the following differences:

#define DEVICE_UDP_SOCKETS 3

This #define is used to override the standard two sockets made available for use with JenNet-IP applications, adding a third socket for use with the 6LoWPAN communications implemented in this device.

Debug Flags

Additional debug flags for the **Protocol.c** module are included.

6.7.3 DeviceProtocol.c

DeviceProtocol.c contains the main source code for the protocol application. It follows the pattern described in Section 6.1.1.3 "Standard Device*Type*.c Features" with the following additions:

6.7.3.1 #includes

Additional #includes are used to provide access to the protocol modules used in **DeviceProtocol.c**.

6.7.3.2 Public Functions

The following public functions are implemented in **DeviceProtocol.c**:

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

The protocol module is initialised with a call to **Protocol_vInit()**.

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

The protocol module main processing is allowed to run each time around the main loop via a call to **Protocol_vMain()**.

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

This callback function is called by the stack for data events at the 6LowPAN level instead of simply calling **Node_v6IpDataEvent()** to discard such messages. As in the other devices, these events are passed on to the protocol module through a call to **Protocol_v6IpDataEvent()** for processing.

```
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 passes these events on to the protocol module through a call to **Protocol_vJipStackEvent()**.

```
void Device_vSecond ( void );
```

This function is called by **Node_vMain()** once per second.

This timer event is passed into the protocol module through a call to **Protocol_vSecond()**.

6.7.4 Protocol.h

Protocol.h contains the public interface to the protocol module.

6.7.4.1 #defines

A small number of #defines are specified here.

`#define PROTOCOL_PORT (1190)`

Specifies the number of the IPv6 port to be opened and used for 6LoWPAN messaging.

`#define PROTOCOL_GROUP 0x1190`

Specifies an IPv6 multicast group address allowing multicast messages at the 6LoWPAN layer.

`#define PROTOCOL_CMD_REG_REQ 'R'`

Specifies the character that indicates a Register Request command within the 6LoWPAN payload.

`#define PROTOCOL_CMD_REG_RSP 'r'`

Specifies the character that indicates a Register Response command within the 6LoWPAN payload.

`#define PROTOCOL_CMD_REG_REQ_SECONDS 15`

Specifies the number of seconds between each node's transmission of the Register Request command.

6.7.4.2 Structures

A small number of structures are declared here:

`typedef struct tsProtocolCmdRegReq;`

Specifies the data in the Register Request command. This structure is used as the payload when sending this command.

`typedef struct tsProtocolCmdRegReq;`

Specifies the data in the Register Response command. This structure is used as the payload when sending this command.

6.7.4.3 Public Function Prototypes

The public function prototypes for the module are defined here. These functions are covered in the next section.

6.7.5 Protocol.c

Protocol.c contains the source code for the protocol handling. It includes code to open the IPv6 port, make a multicast group available for use and send and receive data using the 6LoWPAN layer.

6.7.5.1 #defines

The following #defines are specified:

#define PROTOCOL_CMD_SEND_REG_REQ 0x1

This flag is used to indicate that a Register Request command should be sent to the Coordinator.

6.7.5.2 Local Variables

The following local variables are declared:

bool_t bProtocolSocket;

This flag is used to track whether the IPv6 socket has been opened.

int iProtocolSocket;

This variable contains the socket number after the socket has been opened and is used when sending and receiving communications using the socket.

ts6LP_SockAddr s6lpSockAddrCoord;

Holds the socket address of the Coordinator, uses the short form with an address of 0 that implies messages sent to the address should be directed to the network Coordinator.

ts6LP_SockAddr s6lpSockAddrNetwork;

Holds the socket address of the group used for multicasting to the network.

ts6LP_SockAddr s6lpSockAddrUnicast;

Available for general unicast messaging.

ts6LP_SockAddr s6lpSockAddrLocal;

Hold the socket address of the local device running the software.

uint32 u32CmdRegReqSecond;

Used in Routers and End Devices to track the time of the last Register Request transmission and initiate the next.

uint32 u32CmdSend;

Used in Routers and End Devices to flag which messages need to be sent from the main loop.

6.7.5.3 Public Functions

The following public functions are implemented in **Protocol.c**:

```
void Protocol_vInit ( bool_t bWarmStart );
```

On a cold start, this function initialises the Coordinator, Network and Unicast socket addresses.

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

On the Coordinator the network formation is indicated by the STACK_STARTED event. In this case, the socket is opened with a call to **Protocol_vOpenSocket()**.

On Routers and End Devices the STACK_JOINED event indicates the device has joined a network. In this case, the socket is opened with a call to **Protocol_vOpenSocket()** and the flag set to transmit the Register Request command to the Coordinator.

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

The DATA_RECEIVED event indicates that data has been received.

- First the received message is extracted with a call to **i6LP_RecvFrom()**.

- If successful, the message type is determined from the first character of the payload.

- Where a recognised command has been received, the payload is cast to the appropriate message structure and passed into a message handler for that type of message.

The data for the other types of messages, IP_DATA_RECEIVED and 6LP_ICMP_MESSAGE, are simply discarded with calls to **i6LP_RecvFrom()**.

```
void Protocol_vMain( void );
```

This function is called once each time around the main loop.

On Router and End Device nodes, the flags are checked to see if any messages should be transmitted. If the network is up and the stack is running, the message is sent by calling a command specific function.

On an End Device, if a message needs to be sent, the network is up but the stack is not running, the stack is resumed. On a future call to **Protocol_vMain()** once the stack is running, the command will be sent.

```
void Protocol_vSecond ( uint32 u32TimerSeconds );
```

This function is called once per second.

On Routers and End Devices, the time of the last Register Request transmission is checked against the current time. If it is time to send the command again, the flag is set to trigger this in **Protocol_vMain()**.

6.7.5.4 Private Functions

void Protocol_vOpenSocket (void);

> This function is called whenever the node creates or joins a network.
>
> If the socket has not yet been opened, the following tasks are performed:
>
> - The socket is opened.
>
> - The socket is bound to the node's local IPv6 address.
>
> - The socket is bound to the multicast IPv6 address to allow the use of broadcast packets.

void Protocol_vSendTo (ts6LP_SockAddr   *ps6lpSockAddr,
**   uint8   *pu8Data,**
**   uint16   u16DataLen);**

> This function provides a generic wrapper around the stack's **i6LP_SendTo()** function. Prior to transmitting, it ensures the network is up, the stack is running and allocates a data buffer for the message.

void Protocol_vSendCmdRegRsp (ts6LP_SockAddr *ps6lpSockAddr);

> This function is only included in a Coordinator build. It is used to transmit Register Response command back to a requesting node.

void Protocol_vRecvCmdRegReq (
**   ts6LP_SockAddr   *ps6lpSockAddr,**
**   tsProtocolCmdRegReq *psProtocolCmdRegReq);**

> This function is only included in a Coordinator build. It is used to receive a Register Request command. The Coordinator uses the **Protocol_vSendCmdRegRsp()** function to reply to the sending node.

void Protocol_vSendCmdRegReq (ts6LP_SockAddr *ps6lpSockAddr);

> This function is only included in Router and End Device builds. It is used to transmit the Register Request command to the Coordinator.

void Protocol_vRecvCmdRegRsp (
**   ts6LP_SockAddr   *ps6lpSockAddr,**
**   tsProtocolCmdRegRsp *psProtocolCmdRegRsp);**

> This function is only included in Router and End Device builds. It is used to handle a received Register Response command from the Coordinator.

Appendices

A Revision History – JN-SW-4141 Toolchain

This appendix contains the revision history for the Application Note built on the JN-SW-4141 Beyond Studio for NXP Toolchain, most recent first.

A.1 27/01/2015: Public v2004

Changes

Public release on Beyond Studio for NXP Toolchain.

Lpap443: Devices – detect joining a standalone network via fast commissioning.

Devices detect joining a standalone network via fast commissioning. If the ping interval is 0 the device enters standalone mode (if not already in standalone mode).

Compatibility

Product Type	Part Number	Public Version	Internal Version	Supported Chips
SDK Toolchain	JN-SW-4141	v1111	v1111	JN5168 JN5164
JenNet-IP JN516x SDK Libraries	JN-SW-4165	v1.2	v1107	JN5168 JN5164

A.2 19/12/2014: Internal v2003

Changes

This release includes updated documentation for review and also some small code changes for improved performance.

Lpap362: Template – update documentation for new code structure.

Done.

Lpap365: Routers – choose best setting for u8JNT_IndirectTxBuffers

The number of indirect buffers for packets to End Device children has been increased to improve End Device performance.

Lpap400: Devices – protect against EEPROM corruption caused by noisy power supplies

Two copies of the factory reset data are held in EEPROM to ensure there is always a valid record if one gets corrupted by a write being interrupted by loss of power.

Compatibility

Product Type	Part Number	Public Version	Internal Version	Supported Chips
SDK Toolchain	JN-SW-4141	v1111	v1111	JN5168 JN5164
JenNet-IP JN516x SDK Libraries	JN-SW-4165	v1.2	v1107	JN5168 JN5164

A.3 21/10/2014: Internal v2002

Changes

No changes, built to release alongside JN-AN-1162 JenNet-IP Smart Home Application Note v2002.

Compatibility

Product Type	Part Number	Public Version	Internal Version	Supported Chips
SDK Toolchain	JN-SW-4141	v1111	v1111	JN5168 JN5164
JenNet-IP JN516x SDK Libraries	JN-SW-4165	v1.2	v1107	JN5168 JN5164

A.4 26/09/2014: Internal v2001

Changes

This release tidies up the compilation warnings from the previous release and adds the option to build a Coordinator version of the template device.

Lpap335: Template – Create Coordinator based upon template

DeviceTemplate can be built as a Coordinator that automatically allows any device to join its network. This allows the creation of networks without a Gateway.

Pre-built binaries are *not* provided.

The default Network ID of 0x11111111 should be changed in the makefiles when using this type of Coordinator to prevent it accepting nodes that are trying to join a Gateway JenNet-IP network that should go through the whitelist commissioning process.

Lpap377: Template – Add example of 6LoWPAN layer messaging

DeviceProtocol can be built as a Coordinator, Router or End Device and provides a simple example of opening and using a port for messaging at the 6LoWPAN layer. This allows the use of customer developed protocols instead of or in addition to the standard JenNet-IP MIB and variable protocols. The additional functionality is included in the **Protocol.c** source file which implements a very simple protocol where devices regularly communicate with the Coordinator.

Pre-built binaries are *not* provided.

The default Network ID of 0x11111111 should be changed in the makefiles of when using this type of Coordinator to prevent it accepting nodes that are trying to join a Gateway JenNet-IP network that should go through the whitelist commissioning process.

Lpap378: Devices – Port to Beyond Studio for NXP Toolchain

Done.

Lpap379: Devices – Increase security frame counter on all power cycles

Avoids potential issues with messages being ignored. Previously, this was only increased for devices that were in a standalone mode network.

Lpap382: Template – Coordinator not decrementing authorisation timer

This could result in a node that fails to complete the joining process preventing other nodes joining the network.

Compatibility

Product Type	Part Number	Public Version	Internal Version	Supported Chips
SDK Toolchain	JN-SW-4141	v1111	v1111	JN5168 JN5164
JenNet-IP JN516x SDK Libraries	JN-SW-4165	v1.2	v1107	JN5168 JN5164

A.5 29/08/2014: Internal v2000

Changes

This release is the initial port onto the new Toolchain for internal test purposes. There are a lot of compilation warnings and errors. However the code does compile and run.

Compatibility

Product Type	Part Number	Public Version	Internal Version	Supported Chips
SDK Toolchain	JN-SW-4141	v1051	v1051	JN5168 JN5164
JenNet-IP JN516x SDK Libraries	JN-SW-4165	v1.2	v1107	JN5168 JN5164

B Revision History – JN-SW-4041 Toolchain

This appendix contains the revision history for the Application Note built on the JN-SW-4041 Eclipse Toolchain, most recent first.

B.1 01/08/2014: Internal v1068

Changes

The following application changes were made in this release:

Lpap373: Devices – Better generation of automatic version number

With old scheme, spaces in user names could alter the number of words in the parsed output that generated the version number.

Lpap375: End Devices – Set ping interval to 2

Now End Devices ping when they wake up. Changing this setting prevents pings if application data was sent in the previous wake cycle.

Compatibility

Product Type	Part Number	Public Version	Internal Version	Supported Chips
SDK Toolchain	JN-SW-4041	v1.1	-	JN5168 JN5164
JenNet-IP JN516x SDK Libraries	JN-SW-4065	V1.1.3	v1097	JN5168 JN5164

B.2 23/07/2014: Internal v1067

Changes

This release is built on a new version of the JIP SDK that includes various stack fixes.

The following application changes were made in this release:

Lpsw5263: End Devices – Re-joins with commissioning key instead of network key

If an End Device is power cycled twice without re-joining, the application clears the flag indicating that the device was in a network and so on the second power cycle re-joins using the commissioning key – fixed (in application).

Compatibility

Product Type	Part Number	Public Version	Internal Version	Supported Chips
SDK Toolchain	JN-SW-4041	v1.1	-	JN5168 JN5164
JenNet-IP JN516x SDK Libraries	JN-SW-4065	V1.1.3	v1094	JN5168 JN5164

B.3 20/05/2014: Internal v1066

Changes

This release is identical to the v1065 release with the exception of the binaries being compiled using a patched version of the JenNet library.

The following changes were made in this release:

Lpsw4947: Devices – JenNet only tries to join first scan result entry

Needs to time out the Establish Route Request and move onto the next entry (instead of resetting the stack and starting again) – fixed.

Note this fix requires a patched JenNet library that is not included in the current v1050 installer.

Compatibility

Product Type	Part Number	Public Version	Internal Version	Supported Chips
SDK Toolchain	JN-SW-4041	v1.1	-	JN5168 JN5164
JenNet-IP JN516x SDK Libraries	JN-SW-4065	-	v1050 **(Patched)**	JN5168 JN5164

B.4 08/05/2014: Internal v1065

Changes

This main purpose of this release is to resolve a few final bug fixes.

The following changes were made in this release:

Lpap368: Devices – Use static variable for in call to vJIP_SetDeviceTypes()

Otherwise they don't get correctly reported in the DeviceID MIB – fixed.

Lpap369: Devices – Optional join timeout fires repeatedly

The optional join timeout that stops a device trying to join after a certain time fires repeatedly once the timeout value is reached – fixed.

End Devices should only stop trying to join while the stack is running and the application must place the chip into sleep mode when joining is cancelled – fixed.

Lpap370: Devices – Enable debug earlier

Especially for End Devices, as problems may be caused if not enabled – fixed.

Lpap371: Devices – Make sure MibNwkSecurity_vSecond is called

Instead of calling **MibNwkStatus_vSecond()** twice – fixed.

Lpap372: Devices – Rejoin (without power cycle) scanning all channels and PAN IDs

Need to limit the channel and PAN IDs in the stack when JOINED event is raised – fixed.

Compatibility

Product Type	Part Number	Public Version	Internal Version	Supported Chips
SDK Toolchain	JN-SW-4041	v1.1	-	JN5168 JN5164
JenNet-IP JN516x SDK Libraries	JN-SW-4065	-	v1050	JN5168 JN5164

B.5 16/04/2014: Internal v1064

Changes

This main purpose of this release is to port the applications to the new Mini-MAC to free up additional code-space for application use.

The following changes were made in this release:

Lpap366: End Devices – remove software reset following network loss

Done.

Lpsw4766: End Devices - not sleeping after network re-join

Fixed in stack.

Compatibility

Product Type	Part Number	Public Version	Internal Version	Supported Chips
SDK Toolchain	JN-SW-4041	v1.1	-	JN5168 JN5164
JenNet-IP JN516x SDK Libraries	JN-SW-4065	-	v1050	JN5168 JN5164

B.6 27/03/2014: Internal v1063

Changes

This main purpose of this release is to port the applications to the new Mini-MAC to free up additional code-space for application use.

The following changes were made in this release:

Lpap361: Devices - Adapt for use with Mini-MAC

To create additional code-space for applications.

Known Issues

The following issues remain in this release:

Lpsw4766: End Devices - not sleeping after network re-join

Workaround in place to software reset End Devices when they lose contact with the network.

Compatibility

Product Type	Part Number	Public Version	Internal Version	Supported Chips
SDK Toolchain	JN-SW-4041	v1.1	-	JN5168 JN5164
JenNet-IP JN516x SDK Libraries	JN-SW-4065	-	V1015	JN5168 JN5164

B.7 22/01/2014: Internal v1062

Changes

The main purpose of this release is to optimise the battery life of the End Device template. The Digital Input/Output device has been re-introduced to the package.

The following changes were made in this release:

Lpap279: Devices – Update copyright header

For new devices and year.

Lpap333: DIO MIBs – Adapt for End Device use

DeviceDio and associated MIBs updated to use the new template for use as a Router or End Device.

Lpap356: Devices – Initialise DIO lines for lowest power consumption

Mainly for End Device use.

Compatibility

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

B.8 22/01/2014: Internal v1061

Changes

This main purpose of this release is to optimise the battery life of the End Device template. The End Device Illuminance Sensor has also been added to the package. Only source code for the template device is included in the release package.

The following changes were made in this release:

Lpap341: End Devices – Optimise start-up time

Wait until the stack or UART needs to be used before waiting for the 32MHz clock to stablise.

Lpap342: End Devices – Use network key for rejoin following a power cycle

Instead of the commissioning key.

Lpap350: End Device Template – Exclude MIB Group modules from compilation

End Devices cannot receive broadcasts, so it is pointless to persist group memberships and some code space is gained.

Lpap352: Template – Exclude MIB NodeStatus from compilation

Does not provide much benefit, removed to save code space.

Lpap353: End Devices – Optimise battery life

Removed unnecessary idling, doesn't doze before sleeping, optimised path through main loop.

Respect "stay awake" flag by sleeping then polling a short period later instead of actually staying awake in doze mode.

Alter default timing intervals, 2 seconds parent poll, 5 minutes OND query, 100ms between OND query and data poll.

Compatibility

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

B.9 04/09/2013: Internal v1060

This release re-factors the common MIBs for use with End Devices. The MIB source is now compiled directly into the application rather than into a library.

Note: That the DIO MIBs have not yet been updated to work with End Devices so the source is not included in this release.

Changes

The following changes were made in this release:

Lpap300: MibCommon – remove MIB libraries and re-factor for EndDevice use

The code has been flattened to remove the libraries and the patches to the old JN514x libraries and ROM builds integrated into the flattened code. As a result, this code is now suitable only for JN516x chips.

Lpap336: Template – Update for improved End Device use

Use updated common MIBs.

Allow frequent waking and sleeping for timing purposes without necessarily running the stack. Wake Timer 0 is used to wake and sleep with quite long intervals and is used to run the stack in a normal way. Wake Timer 1 is used to wake and sleep (or interrupt) every 100ms and is used to drive a running timer for application use – the stack is not started when waking from this source. This effectively results in two timer threads running in the application.

Compatibility

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

B.10 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 Semiconductors

For the contact details of your local NXP office or distributor, refer to:

www.nxp.com